Abstract
In game development, procedural content generation (PCG) is an important area of research that aims at generating content algorithmically rather than manually. This paper shows the successful application of a probabilistic grammar in order to procedurally generate a game environment at run-time. The proposed approach helps saving memory, reduces development effort and increases longevity.

1. Introduction

In the context of game development, procedural content generation (PCG) is an increasingly important area of research that aims at generating content algorithmically rather than manually. Traditionally, games are characterized by static content which are precomputed during development. For example, levels are fixed, non-player characters move along predefined paths, objects can be found often in the same places. This has some disadvantages: from the point of view of developers, the effort required during development is time-consuming; on the other hand, from the point of view of players, longevity may be affected. The goal of PCG techniques is to avoid these limitations.

PCG was pioneered in the early ’80s by the game Rogue, in which dungeons to explore are generated randomly. To the same period belongs Elite, a space adventure game in which hundreds of star systems are encapsulated in few tens of kilobytes. A game that makes extensive use of PCG is Minecraft: here, the initial state of the world is mostly random and new areas are generated randomly whenever the player moves towards its boundaries. Recently, PCG has been exploited in combination with affective computing: the aim is to adjust content at run-time according to the user needs and preferences recorded during the game (e.g. [3], [9]).

According to Togelius et al. [7], PCG offers several benefits from three different perspectives: memory consumption, since content can be generated only when needed; development effort, because the expense of manually creating content is alleviated; and longevity, since the same game looks different every time it is played.
However, the application of PCG presents some drawbacks. Since it can result in an unpredictable range of possible game scenarios, there is the need to impose some constraints that allow generating content in a correct and manageable way.

In this paper we explore the application of a probabilistic right-linear grammar \cite{2} in order to procedurally generate the environment of The Ball: Lost in Space\cite{1} game. In particular, the grammar allows generating the path the player character has to go through and, simultaneously, the obstacles to avoid. We show the advantages of employing this technique in terms of reducing development effort and memory usage while increasing the longevity.

The usage of generative grammars in this field has been quite limited, mostly for generating buildings at development-time (e.g. \cite{8}, \cite{5}), or for creating 2D levels at run-time (e.g. \cite{1}). A greater contribution is provided by the so-called L-systems, for example see \cite{6} and \cite{4}; however, the main difference with respect to generative grammars is that rules are applied in parallel rather than once at a time, so resulting in fractal-like structures.

2. The Game

*The Ball: Lost in Space* is an endless running game with a third-person perspective. Like other endless running games, such as *Temple Run*, the player character has to move forward continuously through a path and the main goal is to cover as much distance as possible before inevitably succumbing. Controls are limited to making the character jump, and move left or right, in order to avoid obstacles. These features makes the game suitable for mobile platforms that typically require only a single screen tap to make an action. More precisely, the game consists in controlling the movements of a ball that goes through a wormhole. It ends when the ball crashes into one of the many asteroids that lie on the path or when the ball falls out in the cosmic void. Moreover, it is worth noting that the game includes the task of collecting coins; however, since coins are generated in a totally random way, this feature is not considered. The game was developed with Unity 3D\cite{2}.

The core algorithm of *The Ball: Lost in Space*, aims at generating both the path and the obstacles to avoid, is based on a probabilistic right-linear grammar \cite{2}. Briefly speaking, a generative grammar \(G\) is a quadruple \((T, N, S, R)\), where \(T\) and \(N\) are disjoint finite sets of symbols, called terminal and non-terminal symbols, respectively; \(S \in N\) is a non-terminal symbol called the start symbol; and \(R\) is a finite set of so-called productions. A production is a rewriting rule of the form \(\alpha \rightarrow \beta\), where \(\alpha \in N^+\) and \(\beta \in (N \cup T)^*\). Starting from \(S\), productions are applied iteratively so that non-terminal symbols can be replaced by combinations of non-terminal and terminal symbols, resulting in the generation of a well-formed sentence of the language described by \(G\).

According to the Chomsky hierarchy, a right-linear grammar has all productions in the form \(A \rightarrow AB\) or \(A \rightarrow b\), where \(A\) and \(B\) are non-terminal symbols and \(b\) is a terminal symbol.
A right-linear grammar becomes *probabilistic* by equipping each production with a certain probability of occurring. Obviously, the sum of the probabilities of all the productions with the same non-terminal symbol on the left-hand side must evaluate to 1. The main difference with respect to non-determinism is that in the latter it is deliberately decided to not specify how a certain choice is made.

Using grammars for generating a game environment requires that terminal symbols refer to elements to be drawn on the screen. In our case, terminal symbols refer to the 3D fragments of the wormhole and the asteroids. They are composed in order to generate at run-time a complex and each time different path along which the ball can move. Table 1 shows the correspondence between terminal symbols and the objects to be composed. Note that we have an overall number of thirteen different types of fragments plus one type of asteroid. In Fig. 1 three examples of path fragments are shown. The others are quite similar; they mostly differ from the point of view of the inclination.

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a, b, c, d, e$</td>
<td>Five types of plan</td>
</tr>
<tr>
<td>$f, g, h, i$</td>
<td>Four types of hump</td>
</tr>
<tr>
<td>$l, m, n, o$</td>
<td>Four types of bump</td>
</tr>
<tr>
<td>$z$</td>
<td>Obstacle</td>
</tr>
</tbody>
</table>

Table 1: Terminal symbols of the grammar

![Path fragments](image)

Figure 1: Three examples of path fragments

The grammar allows us managing the pure randomness of the path generation, which would be un gov ern able, by applying some constraints that we have established *a priori*. The constraints can be simply inferred from the rules of the grammar: they allow us to generate paths not too dissimilar from city streets. In particular, they prevent the generation of “unwanted” shapes; for example, steep uphills immediately followed by steep downhills, or steep slopes that suddenly become straight stretches.

Given the 3D models and the constraints for composing them, we can now formalize the grammar as follows: $G = (T, N, S, P)$, with $T = \{a, b, c, d, e, f, g, h, i, l, m, n, o, z\}$, $N = \{A,$
B, C, D, E, F, G, H, I, L, M, N, O, Z, S), and productions and production probabilities in parentheses:

\[
\begin{align*}
S & \rightarrow aA \\
A & \rightarrow aA (0.65) | nN (0.05) | lL (0.05) | gG (0.05) | iI (0.05) | zZ (0.15) \\
B & \rightarrow bB (0.75) | fF (0.25) \\
C & \rightarrow cC (0.75) | hH (0.25) \\
D & \rightarrow dD (0.75) | mM (0.25) \\
E & \rightarrow eE (0.75) | oO (0.25) \\
F & \rightarrow aA (0.7) | iI (0.2) | zZ (0.1) \\
G & \rightarrow dD (0.7) | mM (0.3) \\
H & \rightarrow aA (0.65) | gG (0.05) | iI (0.05) | lL (0.05) | nN (0.05) | zZ (0.15) \\
I & \rightarrow eE (0.7) | oO (0.3) \\
L & \rightarrow bB (0.7) | fF (0.3) \\
M & \rightarrow aA (0.7) | gG (0.07) | iI (0.08) | zZ (0.15) \\
N & \rightarrow hH (0.7) | cC (0.3) \\
O & \rightarrow aA (0.7) | gG (0.07) | iI (0.08) | zZ (0.15) \\
Z & \rightarrow aA.
\end{align*}
\]

Note that the terminal symbol \( z \) formalizes the positioning of an obstacle on the last generated path fragment. For instance, the substring \( aza \) is translated on the screen in a short straight, where is placed an obstacle, followed by another short straight. Moreover, in order to generate a theoretically infinite path, it is worth noting that rules get stuck in an infinite loop. At any moment, only \( n = 100 \) path fragments are stored in memory, and whenever the ball reaches the \( \left( \frac{n}{2} + 1 \right) \)-th fragment, the previous \( \frac{n}{2} \) fragments are removed from memory, and \( \frac{n}{2} \) new fragments are generated.

For what concerns probabilities, it is evident that we preferred flatter paths. In fact, during tests, we realized that assuming equiprobable rules causes the generation of too fluctuating paths, such that the gameplay is negatively affected.

Finally, as regards the amount of obstacles, we opted for an average of ten obstacles each one hundred path fragments, such that the game difficulty can be considered medium.

3. Discussion

Using the classification proposed by Togelius et al. [7], our approach can be defined as: on-line, because content generation is performed at run-time; necessary, because content is required by the player to make progress; parametric, because generation is managed by some constraints; stochastic, since the variation in the outcome between different runs is totally random; constructive, because content is generated and outputted at once without being tested.

Therefore, the proposed approach provides advantages from three different points of views. First of all, from the point of view of the usage of memory, it is worth noting that the path
is generated only when needed. Moreover, since only 100 path fragments are stored at any moment, the path does not need to be kept in memory in all its length from the starting of the execution. Secondly, development effort is reduced: few constraints allow to obtain an incredible number of different combinations in an automatic way. This means that human imagination is augmented and also the effort of composing the fragments together is alleviated. Finally, longevity is increased: since the output is totally random, the player cannot memorize either the path or the positioning of the obstacles. So, he cannot get bored easily.

Grammars are well suited when the content generation requires the composition of few and no complex elements in order to obtain more articulated structures. However, in cases where the elements to be composed are more complex we recognize the difficulty to manage constraints and, so, production rules and their probabilities. Nevertheless, we think probabilistic grammars represent an excellent tool for developing mobile games; in fact, in such a context of use, players typically prefer simple and every time different games.

4. Conclusions

In this paper we have shown The Ball: Lost in Space: an endless running game whose core algorithm is based on a probabilistic right-linear grammar. The grammar allows the algorithmic generation of the environment of the game on the basis of few constraints. The proposed approach helps saving memory, reduces development effort and increases longevity. These features makes it suitable for mobile game development.

An interesting future development of our work could be explore our grammar in combination with affective computing techniques. In this way, it could be possible to monitor players’ emotional states and adjust the path generation at run-time. For example, if the player is bored, the algorithm could make the path more tortuous; vice versa, if the player is nervous, the algorithm could make the path more flat.

Acknowledgements

We wish to thank several people for their contribution to this research: Prof. Fabio Abbattista, whose lectures allowed the development of the game; Giuseppe Scavo, for his help in modeling the path fragments; and Alvin Pastore and Alessandro Bianchi – PhD, for their valuable and constructive suggestions.

References


