Clustering and Finite State Machines for Adaptive Level Generation in Games

Jorge Diz Pico (jorge.diz@estudiante.uam.es), David Camacho (david.camacho@uam.es)

Abstract—**Content generation has been widely used in gaming to offer virtually infinite replayability. Its combination with adaptive techniques also provides a tighter fit between results and gameplay. A new approach for procedural level generation is presented based on a combination of clustering and finite state machines. The method has been designed and deployed as an entry for the Level Generation track of the Mario AI Championship. The implemented architecture is shown to provide great flexibility for both refinement and extension to other games, and the experimental results are shown to prove the validity of this approach.**

I. Introduction

Level generation, in one way or another, has been present since the inception of games history. Initially, its role was to make up for lack of storage memory for levels by creating them on the fly. It's a technique that has featured prominently in many successfully commercial games such as the Diablo saga or Minecraft.

Lately, some of these techniques have been coupled with adaptive algorithms to make the content generated not only random but also tied to the chracteristics of the player. One of the most remarkable is the AI Director found in Left 4 Dead 2, that even goes as far as to adapt the challenges while the level is played if it finds the player low on health or ammo.

In this work, a novel approach to adaptive level generation is presented after testing it on the platform provided by the Mario AI Championship. Previous instances of this competition have featured many worthy techniques; inspired by them, an architecture was devised that can prove not only equally valid but quite flexible for expansion and adaptation as well.

II. Methodology

The presented approach tries to offer a new insight into adaptively generating random levels. It's comprised of two main parts: a data-driven clustering process (mostly offline), and a finite state machine execution (online).

The clustering

Work was started by collecting data from many different players. Each participant was asked to play one of the levels generated by the standard, random algorithm found in the Mario AI Challenge platform, and collect the resulting records.

Social networks and word of mouth were used to recruit volunteers, managing to get 115 different user profiles.

A clustering algorithm was then used to detect the different playstyles and group them. Several methods were tried, such as KMeans or XMeans, but the best results were obtained with Expectation Maximization, when set for three clusters (left by itself, it found five, but of poor quality).

[Some screenshots of Weka comparing how the clusters classify the data, and proving EM is the best fit, would be cool]

One of the clusters was identifiable by the high number of blocks broken and enemies killed, as well as long completion times. The users in this group were nicknamed "explorers". Another cluster was characteristic for its quick completion time, long use of running mode, and disregard for powerups. They were nicknamed "'speeders". Finally, the remaining cluster, presenting a more neutral set of attributes, were nicknamed "ntermediates".

The resulting clusters were then stored for reference and use. This process does not need to be run unless new samples are collected, saving computational effort.

When a new player comes to the game, its records from a previous level are fed to a clusterer, that assigns him to one of the three found clusters. This association is then fed to the next part of the architecture: the finite state machines.

The finite state machines

Each of the clusters has a different automaton associated. This machines have each their own states and transitions defined. This approach allows for a great flexibility on the process given to each user, as different needs and goals are accounted for.

For example, speeders may frown upon blocks of "stairs" because they often force a stop of the running flow. Similarly, explorers tend to enjoy double rows of blocks, an aspect overlooked by the other types of players. Accomodating for different level parts (called "chunks") and their connections yields greater flexibility.

To further enforce modularity, the automata sit on top of a builder layer, offering an API of different level functions. Each state delegates its world-building function to this layer, preventing code repetition on the states and promoting code sharing.

[A diagram of one of the automata, and some explanation of the logic behind the states and their links would be nice.]

III. Conclusions

[A panorama with three different screenshots, one from each automata, spanning both columns, would look good here.]

Records belonging to the three different clusters were fed to the algorithm, and playtest showed that each of the levels generated felt different to the others. [this probably shouldn't be said since we can't prove it... but then, what can we say as results?]

The architecture shown in this paper offers great expansion potential. New states and transitions may be added effortlessly, without need of modifying either the classiffication layer for identifying the player type, or the underlying library of builder functions.

Furthermore, the modelling of the automata after Markov chains allow to state-wide design modifications. For example, a certain state could be made less accessible by applying quite common formulas, without the need to fine tune them by hand. [we should doublecheck this]

Similarly, should the need arise to accomodate a new kind of player, a new automaton can be integrated without interfering with the options already available for current player types.

Future work

The presented architecture can be improved upon in several ways, of which here is suggested the most promising one.

Users are never perfectly encompassed by the cluster they are assigned to; rather, they are more accurately represented by percentages of membership. Rather than force fit a user into a predefined finite state machine, it could be possible to mix and match several of automata until a faithful representation of the user is reached. This could be done by encoding the automata as grammars and using evolutionary algorithms.