Combining Clustering and FSM models for Adaptive Level Generation

Jorge Diz Pico, David Camacho

***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

Procedural 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 soon revealed its potential to make a game "infinite" by presenting the player with a fresh envinroment every time, a quality best exemplified in early 1980s adventure classic *Rogue* (Toy and Wichman). Since then, it has been featured prominently in many commercially successful games such as the roleplaying series *Diablo* (developed by Blizzard, 1996) or the exploration-based *Minecraft* (Mojang, 2009). It's uses have even expanded to terrain creation [1] or quest design [2]. Mario AI Championship Lately, some of these techniques have been coupled with adaptive algorithms to make the content generated not only random but also tied to the characteristics of the player. One of the most remarkable examples is the *"AI Director'"* found in *Left 4 Dead 2* (Valve, 2009), that even goes as far as to adapt the challenges while the level is played if it finds the player low on health or ammunition [3].

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 [4]; 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 is comprised of two main parts: a data-driven clustering process (mostly offline), and a finite state machine execution (online). A third layer, a library of functions that actually build the level, serves as the link between the adaptive generator system and the game API.

Jorge Diz and David Camacho are with the Computer Science Department, Escuela Politécnica Superior, Universidad Autónoma de Madrid. c/ Francisco Tomás y Valiente 11, 28049 Madrid, Spain. (email: jorge.diz@estudiante.uam.es and david.camacho@uam.es)

A. Clustering

The topmost layer undertakes a clustering process on data collected from user play. This approach bases the decisions on empirical metrics and avoids overguessing. This execution can be run offline once and then stored for use afterwards, reducing the computational load on the level generation process. It only needs to be rerun when new samples are collected, to refine and recheck the clustering.

The suggested method for collecting samples is to present a vast array of different players with roughly similar levels (ideally, the same). Having them face comparable challenges makes possible to extrapolate on their gameplay style. So as to not overtrain on a particular player, it is recommended to keep just one significant sample per player. Those collections are then clustered as mentioned before.

Now, when a new player comes to the game, he is made to play a test level in the same vein as the ones used in the aforementioned process. The records of his gameplay are fed to a clusterer that will assign him to one of groups found.

It must be noted that the classification shown here is the most basic approach. Clustering could be further refined, perhaps selecting different parameters at different executions, obtaining the player's position along two different axis; or supporting it with additional inputs like the preferred difficulty. There have been successful experiences working with rhythmic patterns [5] and identifying emotional response from frustation or challenge [6].

B. Finite state machines

Once the player has been identified as belonging to a given group, this association is used to further assign him an automaton. This is done on a one-to-one basis: each group has a finite state machine modeled and tailored to its detected needs, defined upon inspection of their characteristic metrics.

These machines have each their own states and transitions, but most importantly, those transitions are defined as probabilistic. Traversing an automaton is done by picking between the next available states from the current one, weighted by the probability associated to that movement. This feature adds the random component needed to keep the levels generated fresh and unique every time, even though the automata themselves are always the same.

This probabilistic nature in which the next state is only based on the current one makes the automata a model of

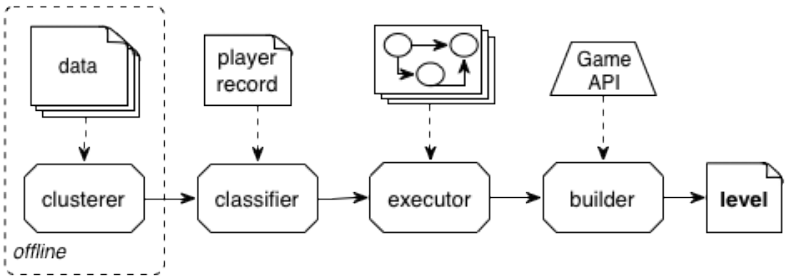

Fig. 1. Layers and modules of the proposed architecture

TABLE I
LOG LIKELIHOOD OF THE DIFFERENT CLUSTERERS

HierarchicalCluster:	79.28
FarthestFirst:	80.70
XMeans:	84.83
SimpleKMeans:	84.03
EM:	86.64

a Markov chain of order one. Statistical considerations can be calculated on the overall probability that each state is reached when the automaton is traversed, opening the door to reducing frequencies in a controlled manner.

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

A diagram of the flow is shown in figure 1.

III. RESULTS

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 Championship platform, and return the resulting records. Social networks and word of mouth were used to recruit volunteers, managing to get 115 different user profiles at the time of this article.

Several clustering algorithms were then tried to detect the different playstyles: *HierarchicalClusterer*, *FarthestFirst*, *XMeans*, *SimpleKMeans* and *EM*. All algorithms, except EM, were wrapped in MakeDensityBasedClusters to compare their log likelihoods, which can be checked above in table I .The best value was obtained with *Expectation-Maximization*, with *SimpleKMeans* and *XMeans* close behind.

The parameters wre configured as standard for the Weka library, except for the number of clusters, that was set to three after several tries. There were two factors to balance: a bigger number meant a better personalization; a small number meant sometimes obtaining clusters with only a handful of players. Four and five clusters were consistently distorting the personality of the clusters (no meaningful conclusions could be extracted from them) and therefore were discarded.

The Mario AI Championship platform stores 45 metrics of user gameplay, mostly related to time spent performing actions and items collected. But some of them were filtered due to the characteristics of the sample level that the platform generates. There are no flower piranhas, no gaps, and no enemies other than goombas. Therefore, all the following metrics were excluded from the study and custering, since their value was always zero: *ArmoredTurtlesKilled*, *CannonBallKilled*, *ChompFlowersKilled*, *GreenTurtlesKilled*, *RedTurtlesKilled*, *enemyKillByKickedShell*, *kickedShells*, *timesofDeathByArmoredTurtle*, *timesofDeathByCannonBall*, *timesofDeathByChompFlower*, *timesofDeathByFallingIntoGap*, *timesofDeathByGreenTurtle*, *timesofDeathByJumpFlower*, *timesofDeathByRedTurtle*.

As mentioned, with three clusters a “personality” for their players could be inferred from their values. One of the them 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 “intermediates”. A full classification of the parameters can be consulted in table II. Figure 2 shows a plot of one of the parameters, *timeRunningLeft* (some jittering was added for easier visualization). This value can be interpreted as a measure of backtracking to interact with missed elements, and is shown here to be quite crucial in telling apart the clusters.

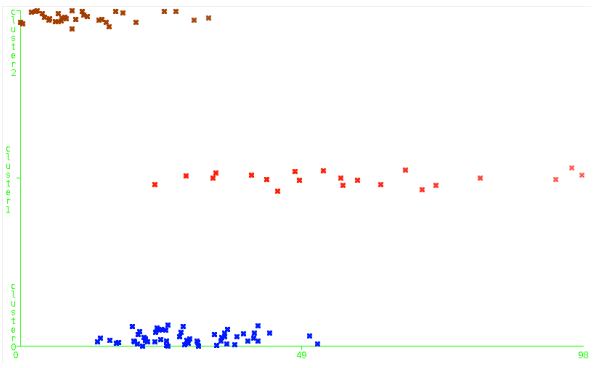

Fig. 2. timeRunningLeft plotted against cluster membership

TABLE II
CLUSTERS AND THEIR PARAMETERS RANKS

Nickname	highest	lowest
"explorers"	GoombasKilled, aimlessJumps, coinBlocksDestroyed, coinsCollected, completionTime, duckNumber, emptyBlocksDestroyed, enemyKillByFire, jumpsNumber, percentageBlocksDestroyed, percentageCoinBlocksDestroyed, percentageEmptyBlocksDestroyed, percentagePowerBlockDestroyed, timeRunningLeft, timeRunningRight, timeSpentDucking, timesPressedRun, timesSwitchingPower, totalEmptyBlocks, totalTime, totalTimeFireMode, totalTimeLargeMode, totalTimeLittleMode, totalPowerBlocks	timeSpentRunning
"speeders"	timeSpentRunning, timesofDeathByGoomba	GoombasKilled, aimlessJumps, coinBlocksDestroyed, coinsCollected, completionTime, duckNumber, emptyBlocksDestroyed, enemyKillByFire, jumpsNumber, percentageBlocksDestroyed, percentageCoinBlocksDestroyed, percentageEmptyBlocksDestroyed, percentagePowerBlockDestroyed, timeRunningLeft, timeRunningRight, timeSpentDucking, timesPressedRun, timesSwitchingPower, totalCoinBlocks, totalCoins, totalEmptyBlocks, totalEnemies, totalTime, totalTimeFireMode, totalTimeLargeMode, totalPowerBlocks
"intermediates"	totalCoinBlocks, totalCoins, totalEnemies	timesofDeathByGoomba, totalTimeLittleMode

Per this observation, and reflecting on the meaning of those user profiles, three automata were designed, attending to the inferred needs of those clusters. 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.

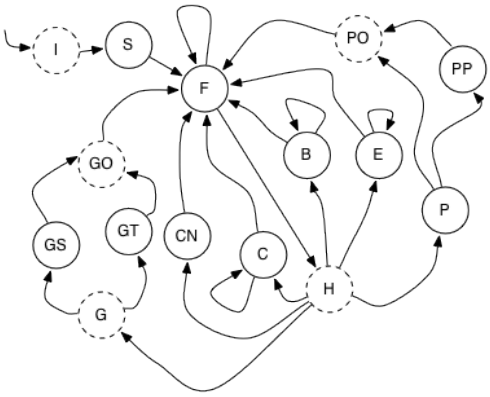

Fig. 3. Simplified automaton designed for intermediates (probabilities removed for clarity)

Figure 3 and figure 4 show a suggested simple automaton for intermediates, with all basic elements present, and a level generated by it, respectively. Transition probabilities are not shown for size constraints and clarity. Dashed states are "empty", that is, they place no elements on the level and exist only for convenience in design.

The general flow of this automaton is to build its initial platform (**S**), some flat land (**F**) and then cyclically go from the global hub (**H**) into branches, each representing a different part of the scene that could be built. At the end

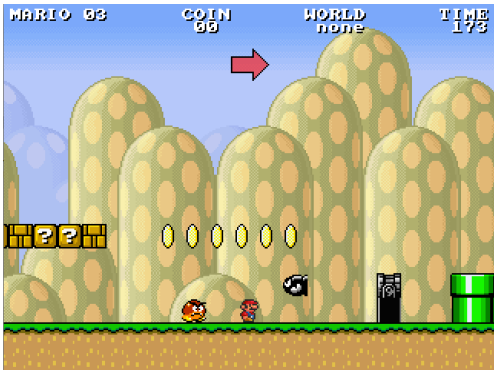

Fig. 4. Screenshot of a random level generated by the above automaton

of each branch, some flat land is added for buffering. Lots of different elements immediately adjacent resulted in not only visually unpleasant, bur rhythmically dissonant segments and severely hurt pacing. Spacing between them helped to offer a smoother cadence.

Let's focus now, for example, on the branch for *gaps*. State **G** serves as entry point for all kinds of gaps, accesible from the hub **H**. This transition provides an opportunity to regulate the overall frequency of gaps in our level. Then, two states bifurcate the way: **GS** places a gap with stairs surrounding it, while **GT** places a regular, bare gap. It's here when the individual probability of each gap can be weighted. Finally, both of them go out through dummy state **GO** (for *gap out*).

In this case **GO** only goes to **F** as mentioned before, but it could be made to loop on itself. This is how blocks (**B**), coins (**C**) or enemies (**E**) are implemented. Each pass through the state places but one sprite on the level. It's the loop that creates a row. Tuning the probability moderates the average number of coins (for instance) that are expected to be found

on a given strike.

IV. Conclusions

Records belonging to the three different clusters were fed to the algorithm, and playtest showed that each of the levels generated had the capacity of feeling different to the others.

The architecture shown in this paper offers great expansion potential. Its modularity expressed through several layers diminishes the interferences of the different parts. For instance, new player types may be detected in the clustering layer, and the changes in the automata layer would be minimal – just adding a new FSM that does not break existing functionality. Similarly, changes in an existing FSM would not affect neither the clustering layer nor the other FSMs of the system. Furthermore, new states and transitions may be added without the need for modifications on the underlying library of builder functions they call upon.

In summary, as different preferences and goals are accounted for, the approach allows for great flexibility on the process followed for each user. Reducing the space of all possible players into a small number of aggregated playstyles also tones down the number of needed automata as to be few enough to be designed by hand. Manually crafting the states and transitions improves the overall arching flow, per the control over sequentially adjacent "chunks" of the level. Albeit keeping its characteristic randomness, the levels are ensured to have an orderly sense instead of being just a messy sequence of pieces.

Future work

One of the most promising aspects of the presented architecture is its flexibility to be enhanced in several ways. We consider combination with evolutionary techniques to be the most rewarding path. Even though determining the existence of transitions (links) between states can be left to the designer's choice, the actual weight of those transitions can be easily obtained by evolution, pondering user feedback from the level play until finding a fine-tuned balance. Similarly, players can be better represented by percentages of similarity to given clusters than by hard assignment. Using those percentages, the automata can be mixed and matched to offer a more personalized experience, by encoding them as grammars and using grammatical evolution.

Acknowledgments

This work has been supported by the Spanish Ministry of Science and Innovation under grant TIN2010-19872 (ABANT).

References

[1] T. Roden and I. Parberry, "From artistry to automation: A structured methodology for procedural content creation," in *Proceedings of the 3rd International Conference on Entertainment Computing*, 2004, pp. 151–156.

[2] A. Tychsen, S. Tosca, and T. Brolund, "Personalizing the player experience in mmorpgs," in *Technologies for Interactive Digital Storytelling and Entertainment*, 2006, vol. 4326, pp. 253–264.

[3] M. Booth, "The AI Systems of Left 4 Dead," in *Artificial Intelligence and Interactive Digital Entertainment Conference (Stanford University)*, 2009.

[4] N. Shaker, J. Togelius, G. N. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi, G. Smith, and R. Baumgarten, "The 2010 Mario AI Championship: Level Generation Track," *In special Issue of IEEE Transactions on Procedural Content Generation*, 2010.

[5] G. Smith, M. Treanor, J. Whitehead, and M. Mateas, "Rhythm-Based Level Generation for 2D Platformers," in *International Conference on Foundations of Digital Games*, 2009.

[6] N. Shaker, G. Yannakakis, and J. Togelius, "Towards automatic personalized content generation for platform games," in *Proceedings of Artificial Intelligence and Interactive Digital Entertainment*, 2010.