Procedural Generation of Balanced Levels for a 3D Paintball Game

Raúl Lara-Cabrera, Víctor Rodríguez-Fernández, Javier Paz-Sedano, and
David Camacho

Department of Computer Engineering
Universidad Autónoma de Madrid, Spain

Abstract. Videogames have become an important field in both the entertainment industry and the computational intelligence research community. Procedural Content Generation (PCG) allows to generate automatically interesting contents for a videogame with a low supervision from the game designers, or even without their supervision. This paper presents a search-based procedural content generator algorithm, that can create interesting maps in a 3D game using an evolutionary approach. Specifically, the algorithm is used in a tactical action game, based on the popular sport Paintball. The maps are generated with an objective in mind: they should be balanced, so any level should not provide an initial advantage of a team over the opponent. The algorithm proposed has been experimentally tested by generating different maps. These maps are represented by graphs of zones and populated with obstacles according to their densities. An evolutionary algorithm evolves these zones looking for an adequate distribution of obstacles in order to generate balanced maps. The experimental analysis shows how the algorithm is able to automatically create suitable maps for the game.

1 Introduction

From its beginning in the early 1970s to the present day, the video game industry has become the main component of the entertainment industry, as well as a powerful international industry capable of penetrating such varied sectors as the economic, social, technological and cultural. According to the Entertainment Software Association [5] there are more than 150 million of gamers just in the United States, who spent 15.4 billion dollars in videogames. These numbers highlight the tremendous impact that videogames have on the economy and society [10], establishing synergies with other fields such as education [1,13], computational intelligence [11] and aerospace industry [21,22].

Making videogames is a time-consuming process, which turns into a high cost, that requires a considerable team of highly proficient professionals from different areas of expertise. For this reason, researchers from the computational and artificial intelligence in games community, have been researching for improvements to reduce both, the resources and time required to design videogames.

Related to this problem, Procedural Content Generation (PCG), which can be defined as: "*generating game content through algorithms instead of manually creating it*", has been an active field within the aforementioned community. The term "content" refers to every component that makes up a video game with an exception: the behaviour of the non-playable characters (NPCs), which makes itself a whole field of research.

The advantages of creating videogames content using algorithms can be summarized as follows [26]:

- It reduces the memory consumption
- It helps the game designer during the process of creating content, thus reducing costs
- It allows for creating novel game-play mechanics
- It facilitates the development of games that adapt themselves to the player
- It may be used as a source of inspiration to the designers

PCG has been used in many commercial games for almost three decades. For instance, Elite [9] is a space trading and combat simulator developed in 1984 with content such as planets and space stations, that is generated procedurally. Traditionally, algorithms have been quite simple as the only goal were to generate playable content, that is, content that fulfil every game-play requirement, so they were practically optimized random number generators. Recent research on PCG techniques aim to create content that is not only correct, but have additional features that may increase the player satisfaction.

One of the most important contents to be generated by using PCG techniques is the map, or level, of a videogame. Almost in every type of videogame, the map can strongly influence the gameplay, and thus, having automatic algorithms and tools to create enjoyable maps is extremely helpful. The research on map generation algorithms have been popular for some years, especially for tactical games with two dimensional scenarios as *Startcraft* [25,27]. However, there is little literature focused on the generation of 3D environments for tactical games, in which the different 3D objects contained in the map and the terrain altitudes are considered to create interesting maps.

In this work, we study the application of PCG techniques into a 3D Action RTS game, named *Paintbol*, in which two teams of characters compete by playing a game based on the popular Paintball sport. Based on Genetic Algorithms [4], our approach tries to create interesting maps for this videogame by modelling the 3D map as a graph of interconnected zones, and then analysing the altitude and level of obstacles (i.e. trees) for each of the zones, which will give some ratings about the zone that will support the optimization process.

The paper is structured as follows: Section 2 analyses the state of the art focusing on level generation, Section 3 provides a description of the game whose maps are procedurally generated by the algorithm described in Section 4. Section 5 describes the experimental setup used to test the feasibility and performance of the generator. Finally, conclusions are drawn in Section 6.

2 Background

Procedural Content Generation (PCG), as part of the field of artificial and computational intelligence in games [12], has become an active research topic as shown by the large number of papers on this topic [15], as well as the creation and growth of conferences and journals that include PCG among their topics.

It is possible to establish several distinctions related to procedures and methods to use when automatically generating content for videogames. According to the taxonomy proposed by Togelius et al. [26], this generation process may be performed at the same time the user is playing (online generation), or prior to the publication of the game during its design and development (offline generation). Moreover, PCG methods should create the content using random seeds in a stochastic manner, using parameters in a deterministic way, as well as a combination of both. At the same time, the content may be necessary (that is, essential for the gameplay) or optional. Depending on the objectives to accomplish, the generation could be done in a constructive way thus ensuring that the content is always valid or using a generate-and-test scheme, in which the content is verified after its creation and discarded if its quality does not meet the standards. Therefore, this paper presents a stochastic PCG method to generate necessary content using a generate-and-test scheme.

As studied in [28], evolutionary algorithms are broadly used when developing PCG algorithms, specially those that follow a generate-and-test scheme as our method do. However, it is possible to use other methods such as cellular automata [17], formal grammars [18] and software agents [3].

The kind of content suitable to be generated by PCG algorithms is varied, being maps and levels the predominant type. For instance, Frade et al. proposed the use of genetic programming to evolve terrains by using both human evaluation as well as quality measures such as the accessibility [7], or the edge length [8] of the terrains. Ferreira and Toledo [6] presented a method able to generate levels for the game Angry Birds. A different approach to tackle the same problem was presented in [23], where Shaker et al. defined a system to tune the parameters that control a level generator for a platform game for adapting it to the way a user plays the game. Another approach is presented by Lara et al. in [19,20], where the authors designed an algorithm to generate balanced and dynamic maps for the real-time strategy game *Planet Wars* using evolutionary computation; in this case the quality of the maps were evaluated by playing and analysing several games between artificial players.

There are additional types of contents that have been created using PCG techniques: Collins [2] explored some approaches to procedural music composition for videogames; Hastings et al. [14], developed an algorithm capable of generate the weapons for a space shooter; and Stockdale [24], who presented a mystery game capable of generating its own narrative.
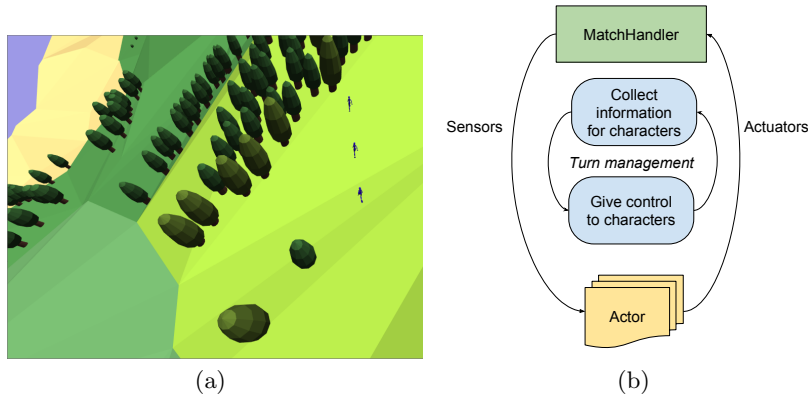
Fig. 1. (a) Screenshot of the game *Paintbol*, showing three agents from the same team following each other and (b) game functional diagram.

3 Paintbol: game description

PaintBol is the name of the videogame used in this work as the basis for our PCG algorithm. It is a tactical action game based on the sport *Paintball*, where two teams of five characters compete by hitting opponents with breakable paintballs in a 3D environment (see Figure 1(a)).

The game's internal basics can be seen in Figure 1(b). The `MatchHandler` class provides the instances of the `Actor` class, which represent the characters of the game, with the information obtained by their senses or sensors (eyes and ears). It also carries out the actions of the characters on the game environment: running, walking, crouching, communicating with other players, shooting, melee-attacking and throwing grenades). Every action in the game produces a certain amount of noise that can be heard by the players if they are close enough to the source of noise when it occurs.

In this game each character has 5 lives. When a character is hit by an attack, he loses one life and disappears for 6 seconds, then reappears in a random place. When the player loses all lives, it goes to the "dead" state and does not reappear.

The scoring system of *Paintbol*, which decides what team has won at the end of a match, considers the following parameters:

- *Lives removed*, representing the number of times that a character has reached an enemy with any possible attack.
- *Melee*, which represents the number of times that a character has reached enemies with a melee attack.
- *Shooting*, which represents the number of times that a character has reached enemies by a gunshot.
- *Grenades*, which represents the number of times that a character has reached enemies with a grenade attack.
- *Stealth*, which represents the number of times that a character has reached enemies before the enemy catches the attacker in its line of sight.

– *Headshots*, which represents the number of times that a character has reached enemies by a gunshot to the head.
– *Multiple attacks*, which represent the number of times that a character managed to hit more than one enemy using one single attack.
– *Friendly fire*, which represents the number of times that a character has reached its own allies with an attack.
– *Remaining lives*, which represents the number of lives of a character at the end of the match.
– *Remaining grenades*, which represents the number of grenades that a character holds at the end of the match.

Then, these parameters are used to compute some **score components**, such as: *Score for lives removed*, *bonus for remaining lives*, *bonus for gunnery*, *bonus for remaining grenades* and *bonus for variety*. Finally, the score of each character is computed as the sum of these score components, and the total score of each team is computed as the sum of scores of each character in the team.

The *3D environment* in which the game takes place, together with the relevance of the topography of the maps and the distribution of the terrain objects with respect to the mechanics of the game, make the terrain analysis (or *terrain reasoning*) an useful and interesting tool for characters in the game.

The behaviour of the teams characters is made up of *heterogeneous* and *independent* agents. They are heterogeneous in the sense that the algorithms that control each of the agents can be completely different, and they are independent because all the agents have private information about the state of the game in function of different parameters like its position and the direction where the character is watching. For example, at the beginning of a match, agents do not know where their allies or their enemies are, or what they are doing unless they are within their arc of vision.

All these features allow the creation of team strategies based on hierarchies, which gives rise to interesting situations when, for example, part of a team is left out of the game and the remaining agents could know how to react to that situation. Agents controlling the characters can communicate partial information about the state of the game, such as sightings of adversaries or allies, locations of strategically advantageous zones in the map. This aspect facilitates and promotes the appearance of emerging behaviours at the level of the teams.

4 A procedural level generator

To facilitate the map generation for the AI platform defined in Section 3 we have designed a search-based procedural level generator capable of evolving interesting levels in a way that any team has no advantage over the others, that is, **balanced levels**. This criterion is interesting for the platform, because the behaviours that could be designed for the teams of NPC players should take the environment into account, to develop their full potential cooperation and collaboration strategies. The core of the level generator is a genetic algorithm that evolves a population of maps through an iterative process. The algorithm has been executed several
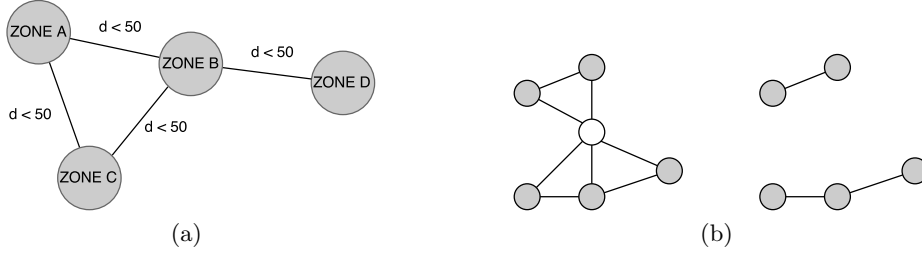
Fig. 2. (a) A map's graph representation. There is an edge between two nodes (zones) if the distance between them is lower than 50 units. (b) Computation of the flanking coefficient of a zone (white node). In this case, after removing the zone, there are 2 connected components, hence $k_i = 1 - \frac{2}{5}$.

times varying some parameters, in order to study its performance and behaviour; the reader can find the results in Section 5.

Regarding map representation, it has been decided to encode a map as a graph that represents a distribution of the zones over the map: nodes are related to the coordinates where the zone is on the map. There is an edge between two nodes if the distance between them is lower than a certain threshold (see Figure 2(a)), which is 50 game units[1] in the engine used to develop the game (i.e. Unity). Every node (zone) has information about its coordinates as well as the density of obstacles on it, while the edges has the density of obstacles between two nodes (zones). In terms of the individuals of the algorithm, they use a vector representation with three consecutive groups of values: the coordinates of the zones, their density of obstacles and the edges' density of obstacles (see Figure 3). Note that there is a density value for every possible edge between two zones but, if those are not close enough, this value is ignored. Due to the gameplay requirements of *Paintbol*, maps must have at most one connected component and the zones must be within the boundaries of the 3D mesh corresponding to the terrain (note that this mesh is an input for the generator).

As stated before, generated maps should be as balanced as possible, in such a way that there is no clear advantage for a team such as an isolated elevated zone with many obstacles. The fitness function is defined as follows:

$$f = \Upsilon(\mu_d) \cdot \alpha + \varsigma(\mu_k) \cdot \beta + \chi(\sigma_d) \cdot \Gamma + \chi(\sigma_k) \cdot \Gamma \tag{1}$$

$$\Upsilon(x) = \frac{e^{-\frac{(x-0.5)^2}{2 \times 0.17}}}{\sqrt{2 \times 0.17 \times \pi}}$$

$$\varsigma(x) = \sin\left(\frac{x\pi - \pi/2}{2}\right) + 0.5$$

$$\chi(x) = \cos\left(\frac{x\pi}{2}\right) + 0.5$$

---

[1] A game unit in Unity is equivalent to 1 meter

where α, β and Γ are coefficients for *defensiveness*, *flanking* and *dispersion* respectively. The main components of the fitness are the mean (μ_d,μ_k) and standard deviation (σ_d,σ_k) of both defensiveness and flanking values of the zones.

$$d_i = \frac{\text{Density}_i + \text{Density}_{paths} + \text{Height}_i}{3} \tag{2}$$

$$\text{Density}_i = \frac{\delta_i}{\delta_{MAX}} \tag{3}$$

$$\text{Density}_{paths} = \frac{\delta_{MAX} - \sum_{j=0}^{\gamma_i} \left(\frac{\delta_j}{\gamma_i} \right)}{\delta_{MAX}} \tag{4}$$

$$\text{Height}_i = \frac{\sum_{j=0}^{\gamma_i} \left(\frac{h_i - h_j}{\gamma_i} \right) + \Delta_{height}}{2\Delta_{height}} \tag{5}$$

The defensiveness of a zone (eq. (2)) is made up of multiple factors:

- The density of the obstacles in that zone (see Eq. (3), with δ_i and δ_{MAX} being the density of node i and the highest density, respectively). Therefore, a zone densely covered with obstacles is considered a high defensive zone.
- The density of obstacles between the zone and the nearest zones (see Eq. (4), where γ_i represents the degree of node i, and δ_j represents the density of the edge j). If a zone has paths with many obstacles its defensiveness should be high.
- A measure related to the height difference between the corresponding zone and the nearest ones (see eq. (5), with Δ_{height} being the maximum difference between the height of two adjacent zones, and h_i,h_j being the heights of the zone i and its adjacent zones j, respectively). Height information comes from the 3D mesh passed as input to the algorithm. The higher the zone is with respect to its neighbours; the more defensiveness score it will have.

$$k_i = \begin{cases} 1 - \frac{\phi_i}{\gamma_i} & \text{if } \gamma_i \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

The flanking coefficient of a zone (node) is computed by counting the number of connected components in the sub-graph made of its adjacent zones (ϕ_i in Eq. (6)) after removing the zone itself (see Fig. 2(b)). If the zone has no connected zone, its flanking coefficient is zero. Hence, if a zone has its neighbours connected between them, it is possible to flank it having defensive coverage while moving between zones, making the game more interesting and dynamic.

Regarding the genetic operators, the generator performs variation with a probability of 0.1, and recombination with a probability of 0.75, by using mutation and crossover operators as follows. The mutation operator applies random permutations to the values of an individual, adding or multiplying a random value to it, drawn from a uniform distribution. The decision of adding or multiplying is made by chance with the same probability. In the event of mutating

an individual in a way that it becomes invalid for the game requirements, the algorithm assigns a fitness of zero to it, to assure that the invalid solution will not be propagated to the next generation due to the elitist selection method. After performing a mutation, the map graph is recomputed to include the possible new edges between two zones if they become close enough due to the mutation.

There are two crossover operators, that are not applied at the same time, but with a probability of 0.5 each. The former is a one-point crossover [16], that selects a random cut point and creates two new individuals with the left slice of one partner and the right slice of the other. The latter is aimed to produce one child, which genes' values are between corresponding genes of parents, and another child, which genes' values are outside of the range formed by corresponding genes of parents. It randomly chooses a factor in the $[0, 1]$ range and then, for each pair of genes, it computes difference value, producing two children: one between and one outside of the range formed by parents genes' values.

5   Experimental Results

To assess the feasibility of procedurally creating balanced levels for the game, we ran the following experiment. The algorithm has been implemented using *AForge.NET*[2] , an open source library focused on computer vision and artificial intelligence written in *C#*. The decision was made taking in account the game was developed using the *Unity3D* engine, which supports the use of *C#* as a scripting language hence making possible integrate it within the game itself.

In order to evaluate the effect of the selection method on the generation process, the next experimental setup was designed: three independent sets of 10 runs of the algorithm, using a different selection method for each one, namely elitism, roulette and rank selection was used. The elite selection method selects a specified amount of best solutions to the next generation. The roulette method selects solutions to the new generation according to their fitness values, so the more fitness value chromosome has, the more chances it has to become member of new generation (each solution can be selected multiple times). As occurs with the roulette selection, the rank selection assigns a probability of being chosen to each solution according to its fitness value; however, the latter orders the solutions by their fitness values and then assigns a probability of 1 to the worst individual, 2 to the next one and so on. The reader can see sample maps that have been created using these selection methods in Figure 4.

---

[2] http://aforgenet.com

Coordinates for zones	Densities for zones	Densities for edges
2N	N	N(N-1)

Fig. 3. Encoding of an individual as a vector of $2N + N + N(N-1)$ values, where N is the number of zones and K the number of edges.

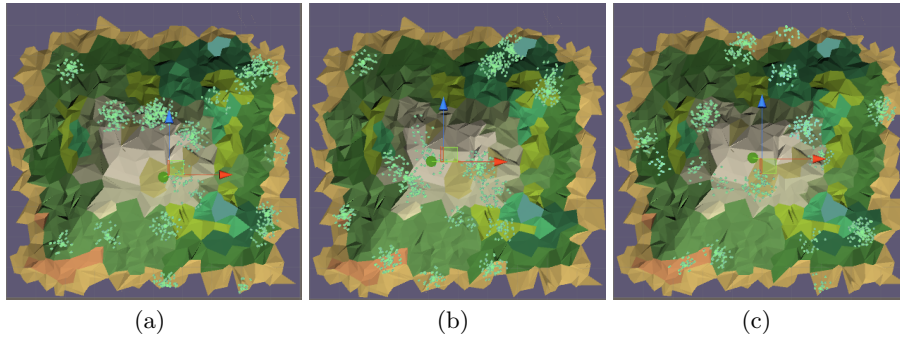(a)   (b)   (c)

Fig. 4. Procedurally generated maps generated using varied selection methods: (a) Roulette, (b) Ranked and (c) Elitist. Light green dots are the obstacles that make the zones, which are distributed according to the generated graph.

Table 1. Algorithm's parameters

Parameter	Value
Number of generations	100
Population size	20
Number of zones	20
Distance between zones (m.)	$[5, 30]$
Density of obstacles	$[50, 100]$

The rest of the parameters were the same for every run as detailed in Table 1. Note there is a fixed number of zones on each map, each one having a density of obstacles that ranges between 50 and 100. Moreover, the distance between those zones is limited to the range $[5, 30]$ to avoid isolated zones.

After running the executions, we have analysed the evolution of the fitness values during the evolutionary process, obtaining the best fitness value for each generation and computing the average over the 10 executions. As can be seen on Figure 5, the roulette-selection method obtains the worst performance, while the others share a barely similar performance according to the standard error of the mean. The deficient performance of the roulette selection might be due to big differences in the fitness values, so those individuals with the lowest fitness are rarely selected. On the contrary, rank and elitist selection exhibit a reliable performance, reaching a fitness of 0.70 (note that the maximum fitness is 1.00). They have a similar beginning and ending with minimal differences in the middle phase of the evolutionary process: elite selection converges faster than rank selection which is, in fact, the expected outcome when using these kind of selection methods (rank selection is more balanced with respect of the exploration and exploitation factors so its convergence is typically slower). All three selection method can generate playable levels with a distribution of zones and obstacle densities that meet the requirements of defensiveness and flanking.
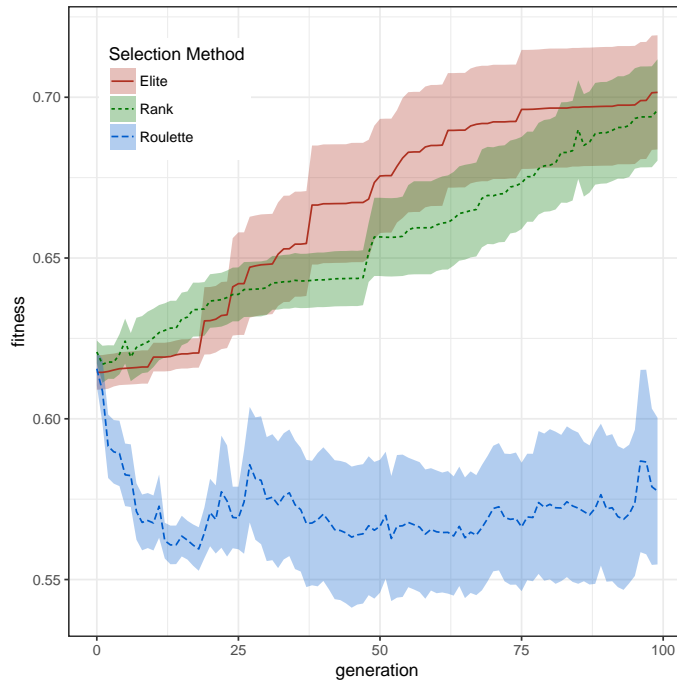
Fig. 5. Evolution of the mean best fitness over the 10 executions using varied selection methods. Shaded area represents the standard error of the mean.

6   Conclusion

In this paper we have introduced an evolutionary algorithm that generates levels for a 3D action game, *Paintbol*, inspired on the Paintball sport. This game has been designed as a research platform for developing intelligent algorithms, which can be applied over teams of agents. To improve its potential, the algorithm evolves graphs that represent zones with obstacles on the map. The distribution of the zones as well as their density of obstacles have been optimized to make balanced maps. As demonstrated by the experimental results, the algorithm is capable of automatically create this kind of maps for *Paintbol*, even using varied selection mechanisms in the algorithm.

According to the obtained results, rank and elitist selection methods perform better than roulette selection (fitness values around 0.7 and 0.57 for the former and the latter, respectively). This is probably due to a wide difference between the fitness values that the population takes during the generational process.

Although the generated maps are fully playable, this paper represents an initial step towards a fully customizable procedural level generator with additional criterion that maps should fulfil making the game more dynamic or even creating aesthetic landscapes. Moreover, it should be possible to test how balanced a map

is by running several games between computer-controlled teams and analysing several game metrics. These are our primary goals to reach in our future work.

Acknowledgment

References

1. Berns, A., Gonzalez-Pardo, A., Camacho, D.: Game-like language learning in 3-d virtual environments. Computers & Education 60(1), 210–220 (2013)
2. Collins, K.: An introduction to procedural music in video games. Contemporary Music Review 28(1), 5–15 (2009)
3. Doran, J., Parberry, I.: Controlled Procedural Terrain Generation Using Software Agents. IEEE Transactions on Computational Intelligence and AI in Games 2(2), 111–119 (jun 2010),
4. Eiben, A.E., Smith, J.E., et al.: Introduction to evolutionary computing, vol. 53. Springer (2003)
5. Entertainment Software Association: Essential facts about the computer and video game industry (2015)
6. Ferreira, L., Toledo, C.: A search-based approach for generating angry birds levels. In: Computational Intelligence and Games (CIG), 2014 IEEE Conference on. pp. 1–8 (2014)
7. Frade, M., Fernandez de Vega, F., Cotta, C.: Breeding Terrains with Genetic Terrain Programming: The Evolution of Terrain Generators. International Journal of Computer Games Technology 2009, 1–13 (2009),
8. Frade, M., de Vega, F.F., Cotta, C.: Evolution of artificial terrains for video games based on obstacles edge length. In: IEEE Congress on Evolutionary Computation. pp. 1–8. IEEE (jul 2010),
9. Frontier: Elite (1984)
10. Gee, J.P., Hawisher, G., Selfe, C.: Gaming lives in the twenty-first century: Literate connections. Springer (2016)
11. González-Pardo, A., Palero, F., Camacho, D.: Micro and Macro Lemmings Simulations Based on Ants Colonies, pp. 337–348. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
12. González-Pardo, A., Palero, F., Camacho, D.: An empirical study on collective intelligence algorithms for video games problem-solving. Computing and Informatics 34(1), 233–253 (2015),

13. Gonzalez-Pardo, A., Rosa, A., Camacho, D.: Behaviour-based identification of student communities in virtual worlds. Computer Science and Information Systems (2014)

14. Hastings, E.J., Guha, R.K., Stanley, K.O.: Automatic content generation in the Galactic Arms Race video game. Computational Intelligence and AI in Games, IEEE Transactions on 1(4), 245–263 (2009)

15. Hendrikx, M., Meijer, S., Van Der Velden, J., Iosup, A.: Procedural content generation for games: A survey. ACM Trans. Multimedia Comput. Commun. Appl. 9(1), 1:1–1:22 (Feb 2013),

16. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. The MIT Press (1992)

17. Johnson, L., Yannakakis, G.N., Togelius, J.: Cellular automata for real-time generation of infinite cave levels. In: Proceedings of the 2010 Workshop on Procedural Content Generation in Games. pp. 10:1–10:4. PCGames '10, ACM, New York, NY, USA (2010),

18. Kelly, G., Mccabe, S.H.: Citygen: An interactive system for procedural city generation. In: In Proceedings of GDTW 2007: The 5th Annual International Conference in Computer Game Design and Technology. pp. 8–16 (2007)

19. Lara-Cabrera, R., Cotta, C., Fernández-Leiva, A.J.: A procedural balanced map generator with self-adaptive complexity for the real-time strategy game planet wars. In: European Conference on the Applications of Evolutionary Computation. pp. 274–283. Springer Berlin Heidelberg (2013)

20. Lara-Cabrera, R., Cotta, C., Fernández-Leiva, A.J.: On balance and dynamism in procedural content generation with self-adaptive evolutionary algorithms. Natural Computing 13(2), 157–168 (2014)

21. Rodríguez-Fernández, V., Menéndez, H.D., Camacho, D.: Design and development of a lightweight multi-uav simulator. In: 2nd IEEE International Conference on Cybernetics, CYBCONF 2015, Gdynia, Poland, June 24-26, 2015. pp. 255–260 (2015),

22. Rodriguez-Fernandez, V., Ramirez-Atencia, C., Camacho, D.: A multi-uav mission planning videogame-based framework for player analysis. In: Evolutionary Computation (CEC), 2015 IEEE Congress on. pp. 1490–1497. IEEE (2015)

23. Shaker, N., Yannakakis, G.N., Togelius, J.: Towards automatic personalized content generation for platform games. In: AIIDE (2010)

24. Stockdale, A.: Cluegen: An exploration of procedural storytelling in the format of murder mystery games. In: Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference (2016)

25. Togelius, J., Preuss, M., Yannakakis, G.N.: Towards multiobjective procedural map generation. In: Proceedings of the 2010 workshop on procedural content generation in games. p. 3. ACM (2010)

26. Togelius, J., Yannakakis, G.N., Stanley, K.O., Browne, C.: Search-based procedural content generation: A taxonomy and survey. IEEE Transactions on Computational Intelligence and AI in Games 3(3), 172–186 (2011)

27. Uriarte, A., Ontanón, S.: Psmage: Balanced map generation for starcraft. In: Computational Intelligence in Games (CIG), 2013 IEEE Conference on. pp. 1–8. IEEE (2013)

28. Yannakakis, G.N., Togelius, J.: A Panorama of Artificial and Computational Intelligence in Games. IEEE Transactions on Computational Intelligence and AI in Games 7(4), 317–335 (dec 2015),