View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by OAR@UM

Evolving Interesting Maps for a First Person Shooter

Luigi Cardamone[1], Georgios N. Yannakakis[2], Julian Togelius[2], and Pier Luca Lanzi[1]

[1] Politecnico Di Milano, Piazza L. da Vinci, 32 - 20133 Milano, Italy
[2] IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen, Denmark
{cardamone,lanzi}@elet.polimi.it, {yannakakis,juto}@itu.dk

Abstract. We address the problem of automatically designing maps for first-person shooter (FPS) games. An efficient solution to this procedural content generation (PCG) problem could allow the design of FPS games of lower development cost with near-infinite replay value and capability to adapt to the skills and preferences of individual players. We propose a search-based solution, where maps are evolved to optimize a fitness function that is based on the players' average fighting time. For that purpose, four different map representations are tested and compared. Results obtained showcase the clear advantage of some representations in generating interesting FPS maps and demonstrate the promise of the approach followed for automatic level design in that game genre.

Keywords: Procedural content generation, Search-based, Evolutionary algorithms, First-person shooters, Player experience, Games

1 Introduction

First-person shooter (FPS) games are video games where the player views the game world from the perspective of the character she is controlling, and where the gameplay involves both navigating into a complex three-dimensional environment and engaging in combat using weapons of different types. FPS games are currently one of the world's most popular video game genres, with no signs of the popularity abating. *Call of Duty*, *Halo*, *Battlefield* and other similar game series sell millions of games each year. Still, developers are plagued by the rising costs of developing content for such games, leading to shorter games and anxiety about creatively and thematically diverging from the mainstream. A typical modern FPS game has less than ten hours worth of single-player campaign and just a few multiplayer maps, despite costing tens of millions of dollars to develop.

Procedural content generation refers to the automatic creation of game content, where "content" is interpreted widely: levels, characters, sound, items, vegetation, and terrain are all viewed as *content* during the development process. While procedural content generation in different forms has been around for more than two decades, it has mostly been applied to peripheral game elements, and usually only in the form of random generation within specific bounds. In contrast,

generation of "core" content such as maps and levels, and generation of content to optimize particular aspects of gameplay or optimize for individual player capabilities, is an active research topic with the results so far considered too immature for commercial games. Recently, a search-based approach to PCG [11], called experience-driven PCG has gained currency [13]. In this approach, evolutionary algorithms or other stochastic search-based optimization algorithms are used to search for content according to some fitness function meant to capture relevant aspects of player experience. Both the representation of game content and the design of the fitness function present interesting research challenges. Some attempts to meet these challenges can be seen in recent work on evolving tracks for racing games [9], rules for board games [1], weapons for space shooters [4], levels for platform games [8] and maps for real-time strategy games [10]. Computational intelligence techniques have been applied to FPS games before, but mostly to create NPC behaviour of various kinds. Some examples are experiments in optimizing the parameters of standard bot scripts [2], imitating the playing style of human players [3] and evolving complete subsumption-based controllers for bots [6]. FPS maps have been subject to analysis by game design researchers, presenting several ideas on design metrics which could be useful for PCG [7], but have to our best knowledge not been automatically synthesized before.

The research described in this paper is novel both in that search-based techniques are used to generate content for an FPS game for the first time, and in that complete playable FPS maps are generated for the first time. The paper is structured as follows: in the next section, we present the *Cube 2* game engine which will be used for the experiments. We then present our general approach to evolving FPS maps, including the fitness function. This is followed by descriptions of the various map representations, and of the experiments we carry out to compare their suitability for evolving playable maps. A concluding section discusses the next steps to take in order to refine this technique.

2 Cube 2

Cube 2: Sauerbraten [12] is a free open-source FPS game that supports both single- and multi-player gameplay. Cube 2 comes with several graphical character models, a set of weapons and a large number of maps. The engine underlying Cube 2 is fast and sophisticated, allowing smooth rendering of high-polygon environments, approximating the visuals found in commercial games (see Figure 1). Technically, the game engine is based on a 6 directional height-field in octree world structure which also supports lightmap-based lighting with accurate shadows, dynamic shaders, particles and volumetric explosions. Cube 2 also supports a simple but complete configuration/scripting language which allows customization of all menus and most other aspects of the game, and which makes it easy to build "mods" of the game.

One of the standout features of the engine is the capability for in-game geometry editing and also multi-player cooperative editing. The integrated map

Fig. 1. Screenshots from Cube 2: Sauerbraten

editor is very powerful: it allows the user to build and edit new shapes, to apply textures and to add objects of various materials (such as water, lava and glass), and to add light sources, among several other actions. Since Cube 2 is open source, the map editor itself can be modified and extended in whatever way necessary. This feature is crucial for our purposes, as we need to inject evolved maps back into the game, and it is one of the main reasons we chose to use Cube 2 rather than a better-known commercial FPS game. The other main reason is that the game engine allows us to run the game in "headless" mode, i.e. without visualization, where it can be speed up to run as fast as the processor permits.

3 Evolving Maps for an FPS Game

In this paper we apply evolutionary algorithms to evolve maps for a multi-player FPS game. A multi-player FPS is a game where several players (humans and/or bots) fight on the same map. Several game modes are possible with differences in rules; in the most basic mode, "deathmatch", the rules are simple: when a player character is killed, it will spawn in another point of the map after a few seconds. The game terminates after a fixed amount of time, and the player with most frags (i.e. the player which killed more opponents) wins the game.

A FPS map usually consists of a series of rooms and corridors plus several spawn-points and resource items (either weapons or bonuses). Maps may have several different levels with floors above and below each other, and features such as stairs, ramps and elevators for moving vertically between floors. In this work we will focus on maps with only a single floor.

The goal of this work is to evolve maps with potential for interesting gameplay. It is generally accepted that some FPS maps allow for more interesting and deeper gameplay than others, for example by rewarding skillful use of complex tactics, and by forcing players to vary their tactics so that they cannot use the same patent trick all the time to win. It is generally accepted that such maps are of better quality than others. Indeed, much work goes into exquisite balancing

of those maps that are available as paid downloads for popular FPS games, to make sure that no single strategy will dominate the map. For brevity, we will in the following refer to maps with potential for interesting gameplay as *promising* maps.

3.1 Fitness function

Naturally, it is hard to design an accurate estimator of the promise of a map, as this will require predicting the preferences of a great number of players (e.g. as in [8]). In the future, we plan to devise more complex fitness functions based on data-driven modelling of player capabilities and preferences. For now, we will settle on a simple theory-driven fitness function.

We assume that the promise of a map is directly linked to the *fighting time of the player*, which is defined as the duration from the moment in which the player starts to fight an opponent to the moment in which the player is killed. Since during an entire match the player will die and spawn multiple times, we can compute the average fighting time value for the game per player, T_f. A small T_f value means that a player gets killed very quickly after starting to fight an opponent. If the T_f value is large it means that after the player first gets damage in a fight, she survives and can prolong the fight because the map affords possibilities to escape, to hide, to find health bonuses or better weapons. Since an FPS map presents several features that, from a strategic point of view, can be exploited to engage the player in longer and more interesting fights the T_f value appears to be a good measure of the promise of a map.

The best way to assess the T_f value of a map would be to play several matches with human players on that map and collect the statistics, yielding an *interactive* fitness function according to the classification in [11]. Unfortunately, it is not practical to use human players in our experiments because it would require them to play thousands of matches (in the future, this might be a possibility using massively multiplayer games, similar to the approach taken in [4]). Instead we simulated matches of 10 minutes among 4 bots and we measured the average T_f value across all bots, $\overline{T_f}$, yielding a *simulation-based* fitness function according to the same classification.

The complete fitness function has another component in addition to the T_f value: the *free space of the map*, S. We explicitly want our fitness to promote the generation of larger maps since very small maps do not leave enough space for the placement of weapons and spawn-points, leading to unrealistic values of T_f. It is worth mentioning that the maximum size of a map is bounded and that in the best maps generated (see Section 5) S contributes to less than 20% of the total fitness value. Given the above, the complete fitness function of a map is as follows:

$$f = \overline{T_f} + S \tag{1}$$

where $\overline{T_f}$ is measured in milliseconds and it is an integer greater than 0, and S represents the number of free cells in the map and is bounded between 0 and 4096.

To evaluate maps using a simulated match we had to address two main points: generation of way-points and acceleration of the game. While for the latter the solution was as simple as disabling the graphical rendering (enabling the simulation of a 10 minutes match in about 10 seconds using a computer with an 3.00 Ghz Intel Core 2 Duo processor), for the first we had to implement a way-point placement algorithm. The bots available in Cube 2, like the ones in many commercial games, depend on a list of way-points to navigate in a given map. On that basis, points are represented in a graph on which bots apply the usual A^* algorithm to move between bonuses, weapons and enemies. Therefore, it was necessary to generate the way-points on the fly for each new map. Unfortunately, the game gives no support for automatic generation of way-points as they are usually placed by the human designer of the map. To overcome this problem we implemented an algorithm for way-point generation that follows these steps: (i) compute the free cells of the map; (ii) place a way-point on every free cell; (iii) connect each way-point with its four neighbors (up, down, left, right) if there are no obstacles between them; (iv) align every resource on the map to the grid of the way-points.

4 Map Representations

After the fitness function, the other important design choice in search-based PCG is how to represent the content. The representation must allow for the expression of a wide range of interesting content, but also enable the search algorithm to easily find content with high fitness. In this paper, we propose and compare four different representations for FPS maps.

First, we need to distinguish between the *genotype* and *phenotype* for these maps. The phenotype is isomorphic to an actual map in the game. For all map representations, the phenotype structure is the same: a matrix of 64×64 cells. Each cell is a small piece of the map with a size suitable to contain a player character. Each cell can be either a free space or a wall (with a fixed height). Each free space can be empty or can contain a spawning point or a resource item (weapon or health bonus). The phenotype is saved in a text file and loaded in the game for playing using a specific loader that we implemented in Cube 2.

The structure of the genotype, on the other hand, is different for each representation. The genotype is what is evolved by the genetic algorithm. Each representation comes with a procedure for constructing a phenotype from the genotype; the simpler this procedure is, the more *direct* the representation is said to be. When a genotype is evaluated, the following happens:

1. the genotype is used to build the phenotype;
2. then the phenotype yields a specific map for Cube 2 and a simulated match starts in that map;
3. the statistics collected during the match are used to compute the fitness.

The four representations are described below in order of decreasing directness.

The most direct representation is named **Grid** and assumes that the initial configuration of the map is a grid of walls. In particular a 9 by 9 grid is used to divide the map into 81 squares. Each of the wall segments of a square can be removed to create rooms and corridors. The genome represents which of these wall segments are active (on) or not (off). According to the **Grid** representation scheme each gene encodes the state of two wall segments of the cell: the one on the top and the one on the right. Thus, each gene can take four possible values: 0, if both wall segments are off; 1, if only the top wall segment is on; 2, if only the right wall segment is on; and 3, if both segments are on.

The second, less direct representation is named **All-White**. It assumes that the initial map is empty (with walls only at the borders) and searches for fit maps by adding wall elements on the empty space. The genome encodes a number of wall blocks, N_w (N_w equals 30 in this paper), each represented by three values, $< x, y, l >$, where x and y define the position of the top-left corner of the wall and l represents the length of the wall. If $l > 0$, the resulting wall is aligned to the x-axis; otherwise it is aligned to the y-axis. According to this representation, the width of each wall block equals 1.

The third representation is named **All-Black** and is, in a sense, the exact opposite to the **All-White** representation. This representation starts with an initial map full of wall blocks, and gradually adds free spaces in search of fitter maps. The genome encodes a number of free spaces of two types: the *arenas* and the *corridors*. Arenas are square spaces defined by the triplet $< x, y, s >$, where x and y represent the coordinates of the center of the arena and s represents the size of it. Corridors are rectangular-shaped free spaces with width fixed to 3 cells. Corridors are encoded in the same way as wall blocks via the three values of $< x, y, l >$. In the experiments presented in this paper, the **All-Black** representation builds on 30 corridors and 5 arenas.

The most indirect representation is called **Random-Digger** and as the **All-Black** representation, it builds maps on an initial map configuration which is full of wall blocks. The genome encodes the policy of a very simple agent that moves around and frees up the space in every cell it visits, in a way reminiscent of turtle graphics. The resulting map is a trace of the path the agent followed for a fixed number of steps. The agent starts at the center of the map and its policy is represented by four probability values: $< p_f, p_r, p_l, p_v >$ where p_f is the probability of going forward along the current direction; p_r is the probability of turning right; p_l is the probability of turning left; and p_v is the probability of visiting an already visited cell. The last probability is very important since it controls the exploration rate of the digger.

The first three representations can generate maps with some parts that are not reachable from the all other parts of the map. There are two main approaches to overcome this problem. The first approach attempts to repair the map so that it becomes fully connected. This solution has several drawbacks: it is complex to implement, it can be computationally expensive and it may heavily modify the initial shape of the map. The second approach focuses on simply removing the unreachable parts from the final map. In this paper we follow the second

approach by identifying all cells that were reachable from the center-point of the map and then remove all cells that are not reachable.

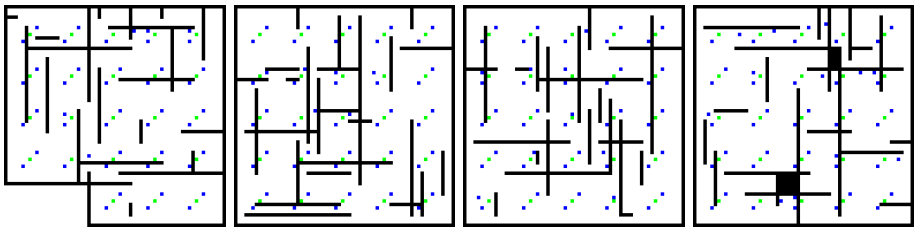
Before a complete Cube 2 map can be generated from the phenotype, we need to add spawning points, weapons and health bonuses. We do this through a simple uniformly-distributed heuristic as follows: (i) the matrix of the phenotype is divided into squared blocks; (ii) for each block the number of free cells is computed; (ii) if this number is bigger than a given threshold two spawn-points and one resource item (a weapon or a health bonus) are placed inside the block.

5 Experiments

To evolve the maps, we applied a simple genetic algorithm with standard operators: tournament selection with tournament size 2, single point crossover, mutation rate of $\frac{1}{n}$ (where n is the length of the genome), mutation range 0.2 (following a uniform distribution) and a crossover probability of 0.2. The parameters of the genetic algorithm were set empirically. We applied the genetic algorithm with a population size of 50 individuals and let it run for 50 generations. Each evolutionary run took approximately 6 hours to completion and the experiment was repeated 10 times for each representation.

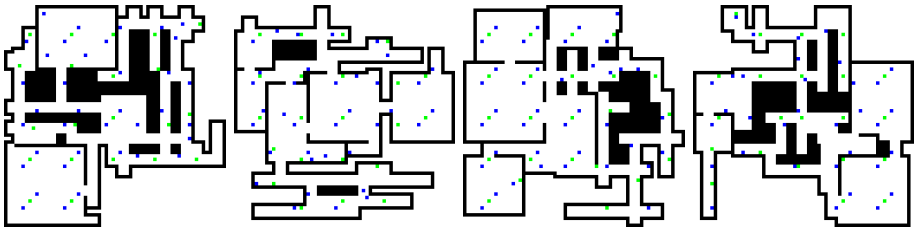
Figures 2, 3, 4 and 5 display the four highest performing maps evolved for each representation. The 2D image of each map illustrates walls as black segments, spawn-points as blue dots, and resource items (either weapons or health bonus) as green dots. Figure 6 illustrates how one of the best map evolved appear rendered in 3D when they are loaded in Cube-2. To test for significance, we run a t-test on the highest performances obtained via the 10 GA runs among all representations. According to the results: (i) **All-White** generates maps which are statistically better than **All-Black** and **Random-Digger** (p-value < 0.001 in both comparisons); (ii) **All-White** evolved better maps than **Grid** but the difference is not statistically significant (p-value $= 0.141$); (iii) **Grid** maps are statistically better than **All-Black** and **Random-Digger** maps (p-value < 0.001 in both comparisons); and (iv) the maps generated with **All-Black** are statistically better than the **Random-Digger** maps (p-value < 0.001). In addition, we performed experiments to test for the sensitivity of the fitness value by evaluating a map multiple times. Even though the variance of the fitness can be rather large and dependent on the map structure, the initial placement of bots and weapons and bot behavior the fitness order among the four representations is maintained and the statistical effects hold.

As can be seen from the results obtained, all the four representations are able to generate playable maps. Each representation allows the emergence of some peculiar features that strongly characterize the evolved maps. The **Random-Digger** representation generates maps with many long corridors and few small arenas. The **All-Blacks** representation instead, generates several bigger arenas while corridors are usually shorter and may present dead ends. The **Grid** representation generates very interesting maps with a high degree of symmetry. Finally, the **All-White** representation generates the best maps according to the



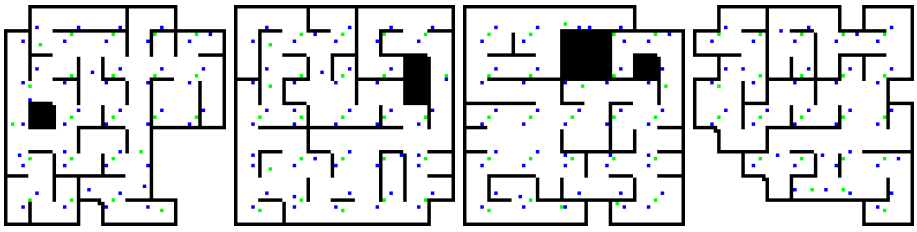
(a) $f = 20705$ (b) $f = 21146$ (c) $f = 21931$ (d) $f = 22882$

Fig. 2. Best Maps Evolved using Representation **All-White**



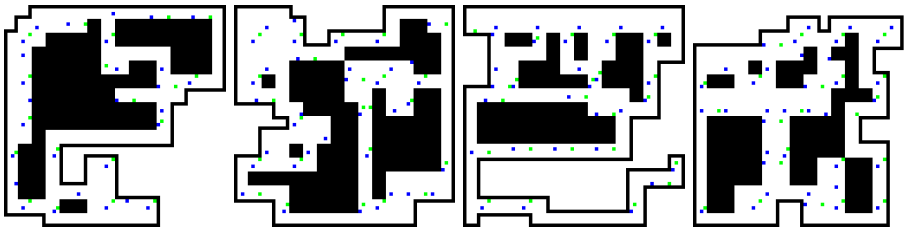
(a) $f = 14520$ (b) $f = 14530$ (c) $f = 15410$ (d) $f = 15801$

Fig. 3. Best Maps Evolved using Representation **All-Black**



(a) $f = 17282$ (b) $f = 18028$ (c) $f = 20812$ (d) $f = 22520$

Fig. 4. Best Maps Evolved using Representation **Grid**



(a) $f = 12500$ (b) $f = 12964$ (c) $f = 12972$ (d) $f = 13202$

Fig. 5. Best Maps Evolved using Representation **Random-Digger**

(a) 3D map overview (b) First-person perspective

Fig. 6. One of the best evolved map ($f = 21931$) loaded in the game *Cube 2*.

considered fitness function. The high fitness values of **All-White** maps are justified by the coupling of many narrow passages with big arenas which generate many small spaces for a player to hide and trap its opponent or pick health bonuses after a fight.

It is worth noticing that the 2D top-down map images of Figures 2, 3, 4 and 5 may be misleading for some readers. For instance, the **All-White** maps are less symmetrical and aesthetically pleasing than the maps of the **Grid** representation; thus, one may infer the inferiority of the **All-White** maps with respect to their gameplay value. However, this aesthetic notion is reverted once the map is viewed and played from a first person perspective (see Figure 6) as it is confirmed by our preliminary results from a user study.

6 Conclusions and Future Work

We have devised four different representations for first-person shooter maps, and a fitness function for evaluating their potential for interesting gameplay. We have also performed several experiments to verify that we can evolve playable FPS maps using these representations and the fitness function. Further, we have used the fitness function to compare the promise of maps evolved with each of the representation. From our results, it seems that the **All-White** and **Grid** representations have clear advantages in this respect.

Several legitimate objections can be raised against our fitness function, and each of them suggest avenues for future work. One objection is that it depends on the control logic of the default bots in Cube 2. This control logic has not been presented in the paper, and is indeed not entirely transparent from the source code. Using custom-designed bots would permit us to tune our fitness function in more detail, and explore new fitness functions related to the present one, such as the performance of a numerically or tactically superior team against an inferior. Adjusting the bots' behaviour to match human gameplay styles might

also improve the fitness function; clues to as how to do this might be taken from the submissions to the 2k BotPrize competition [5]. Another objection is that we have not yet validated the efficacy of our fitness function with user studies, and can therefore not claim that our measure of potential for interesting gameplay corresponds with human players' judgments. User studies are currently our top priority, and our preliminary results suggest that players do prefer maps with higher fitness. Following the principles presented in [13, 8] we also plan to study the effect of map design on several emotional states of players, and synthesize models of player experience from player behavior and FPS map design. These models can then be used to construct adaptive FPS games, that create appropriate maps to match the skills and preferences for particular players (or groups of players) so as to create an optimally engaging gameplay experience.

References

1. Browne, C.: Automatic generation and evaluation of recombination games. Ph.D. thesis, Queensland University of Technology (2008)
2. Cole, N., Louis, S.J., Miles, C.: Using a genetic algorithm to tune first-person shooter bots. In: Proceedings of the IEEE Congress on Evolutionary Computation. pp. 139–145 (2004)
3. Gorman, B., Thurau, C., Bauckhage, C., Humphrys, M.: Believability testing and bayesian imitation in interactive computer games. In: Proceedings of the Conference on Simulation of Adaptive Behavior (SAB) (2006)
4. Hastings, E.J., Guha, R.K., Stanley, K.O.: Evolving content in the galactic arms race video game. In: Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on. pp. 241–248 (Sept 2009)
5. Hingston, P.: A new design for a turing test for bots. In: Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG) (2010)
6. van Hoorn, N., Togelius, J., Schmidhuber, J.: Hierarchical controller learning in a first-person shooter. In: Proceedings of the Symposium on Computational Intelligence and Games (CIG) (2009)
7. Hullett, K., Whitehead, J.: Design patterns in fps levels. In: FDG '10: Proceedings of the Fifth International Conference on the Foundations of Digital Games. pp. 78–85. ACM, New York, NY, USA (2010)
8. Pedersen, C., Togelius, J., Yannakakis, G.N.: Modeling Player Experience for Content Creation. IEEE Transactions on Computational Intelligence and AI in Games 2(1), 54–67 (2010)
9. Togelius, J., De Nardi, R., Lucas, S.M.: Towards automatic personalised content creation in racing games. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG) (2007)
10. Togelius, J., Preuss, M., Beume, N., Wessing, S., Hagelbäck, J., Yannakakis, G.N.: Multiobjective exploration of the starcraft map space. In: Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG). pp. 265–272 (2010)
11. Togelius, J., Yannakakis, G.N., Stanley, K.O., Browne, C.: Search-based procedural content generation. In: Proceedings of EvoApplications. vol. 6024. Springer LNCS (2010)
12. V/A: Cube 2: Sauerbraten. http://sauerbraten.org/
13. Yannakakis, G.N., Togelius, J.: Experience-driven Procedural Content Generation. IEEE Transactions on Affective Computing (2011)

View publication stats