Proceedings, The Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-15)

Multi-Level Evolution of Shooter Levels

William Cachia, Antonios Liapis, Georgios N. Yannakakis
Institute of Digital Games, University of Malta, 2080 Msida, Malta
{william.cachia.07, antonios.liapis, georgios.yannakakis}@um.edu.mt

Abstract

This paper introduces a search-based generative process for first person shooter levels. Genetic algorithms evolve the level's architecture and the placement of powerups and player spawnpoints, generating levels with one floor or two floors. The evaluation of generated levels combines metrics collected from simulations of artificial agents competing in the level and theory-based heuristics targeting general level design patterns. Both simulation-based and theory-driven evaluations target player balance and exploration, while resulting levels emergently exhibit several popular design patters of shooter levels.

Procedural content generation (PCG) has often been used in the game industry to create single-player experiences such as the dungeons of *Diablo* (Blizzard 1996), or large-scale gameworlds which do not impact player balance (or where imbalance can be mitigated via e.g. terraforming) in games such as *Endless Space* (Amplitude 2012). Competitive games often used for electronic sport, such as *Starcraft* (Blizzard 1996) or *Call Of Duty: Advanced Warfare* (Activision 2014), rarely include procedurally generated content; this can be traced back to a variety of design decisions on the core gameplay (which often involves memorization of locales and action sequences) and production choices (e.g. balancing teams through matchmaking rather than generated levels of varying challenge).

While the game industry continues to focus on the generation of levels for single-player games, recent academic interest has been targeting genres traditionally considered off-limits to PCG. Levels for strategy games such as *Starcraft* were evolved by Togelius et al. (2013), who requested experienced players' feedback on the best generated levels. Experts' responses highlighted issues with human perception of balance, which in turn opens up new research on reducing such effects via e.g. hard-coded symmetry (Uriarte and Ontañon 2013). For first person shooter (FPS) games, stochastic search has been used to generate simple, flat levels based on simulations with artificial agents (Cardamone et al. 2011; Lanzi, Loiacono, and Stucchi 2014) or on human votes (Ølsted, Ma, and Risi 2015); however, search-based PCG methods of FPS levels remains an under-explored area.

This paper introduces a method for evolving FPS levels with more than one floors, allowing for more divergent gameplay. A second floor allows popular design patterns of FPS levels to emerge naturally, including multiple flanking routes, positional advantage and sniper locations (Hullett and Whitehead 2010). The design of the algorithm is largely inspired by Cardamone et al. (2011), but expanded to also optimize the placement of weapons, healthpacks and spawnpoints. The larger search space (a result of a more complex, multi-floor level) needs to be explored in an efficient manner; the proposed method involves a two-stage optimization process where the level architecture is evolved in the first stage while the game object placement is evolved in a second stage. In order to evaluate the complex levels, several simulation-based heuristics from Cardamone et al. (2011) and Lanzi, Loiacono, and Stucchi (2014) are combined with theory-driven evaluations of popular game level design patterns (Liapis, Yannakakis, and Togelius 2013). Each stage of evolution uses only those heuristics appropriate to that stage, alleviating optimization effort (to a degree). Results show how shooter levels with two floors compare to those with one floor, based on the heuristics' scores but also based on the gameplay of the artificial agents which evaluate them.

Related Work

This paper discusses a method for generating shooter levels via a genetic algorithm. The genetic algorithm attempts to optimize a simulation-based fitness function derived from playthroughs with artificial agents, as well as fitness functions which capture (implicitly or explicitly) design patterns common in shooter levels. The sections below elaborate on the different methods of generating game levels procedurally, and present an overview of level design patterns.

Procedural Content Generation and Shooter Levels

There is a long history of procedural content generation in the game industry; most generators in commercial games target levels, from the dungeon generator of *Rogue* (Toy and Wichman 1980) to the world generator of *Civilization V* (Firaxis Games 2010). Recent academic interest in PCG (Togelius et al. 2011) attempts to identify new types of game content which can be generated (Togelius, Nelson, and Liapis 2014), but also to discover new methods for generating it. Most generative methods fall under the categories of

constructive, generate-and-test and search-based. Constructive methods use finely tuned algorithms to generate game artifacts without testing whether they are playable or of high quality; generate-and-test methods perform such tests and regenerate artifacts until playability constraints are met. Search-based methods often include some form of stochastic search (and specifically evolutionary computation) and attempt to optimize one or more desirable qualities in the artifacts. Quality can be derived from mathematically defined fitness functions grounded in theoretical frameworks, from playtraces of simulated games (*simulation-based* evaluation), from models of player experience learned from real human traces (Yannakakis and Togelius 2011), or directly from human interaction (Liapis, Smith, and Shaker 2015). The generator described in this paper uses both simulation-based evaluations and mathematically defined fitness functions which build on popular level design patterns.

One of the main inspirations for the proposed generator is the work of Cardamone et al. (2011), one of the first attempts at evolving FPS levels. In their paper, Cardamone et al. experiment with different representations (including the *digger* and *all-black* approach used in this paper) of shooter levels as genes, and use simulation-based evaluations to drive the evolutionary process. Their approach optimizes the architecture based on the space available for navigation and the average fighting time of artificial agents in a playthrough; powerups (healthpacks, weapons) and player spawnpoints are dispersed deterministically and are not evolved. While this paper uses the fitness dimensions of Cardamone et al. (among other metrics), the proposed two-stage approach can optimize game object placement after the level architecture is evolved. The dual representation of level architecture and game object placement used in the two-stage evolutionary process resembles that of Cook and Colton (2011) which used a similar composite representation (including the game's ruleset in the evolvable genes) but evolved all components simultaneously rather than in stages.

Design Patterns and Shooter Levels

Design patterns have been used in many different fields (software architecture being an obvious example) to describe a common problem in the field and then describe "the core of a solution to that problem" (Alexander, Ishikawa, and Silverstein 1977). Björk and Holopainen (2004) introduce *game design patterns* as "a tool for understanding and creating games". The numerous game design patterns of Björk and Holopainen were distilled to those most relevant to level design (and most straightforward to compute) and reworked as fitness functions by Liapis, Yannakakis, and Togelius (2013) for the evolution of game levels. As the patterns which inspired them were general across games, so do these evaluations of *exploration*, *area control* and *balance* apply to many game genres, albeit at the cost of specificity. This paper combines such general evaluations of game level patterns with simulation-based evaluations (which are specific to the game genre, to the game engine and to the opponent AI in the game) in order to drive the evolution of object placement in a multi-floor shooter level.

Although the design patterns of Björk and Holopainen can be used to describe all games, design patterns specific to shooter levels have also been identified. Hullett and Whitehead (2010) identify ten patterns pertaining to positional advantage (*sniper location*, *gallery*, *choke point*), large-scale combat (*arena*, *stronghold*), alternate gameplay (*turret*, *vehicle section*) and alternate routes (*split level*, *flanking route*). Not all shooter levels need to include all patterns (especially e.g. patterns of alternate gameplay). While the generator does not explicitly evaluate evolving levels based on these patterns, several design decisions in its conception are informed by the patterns of Hullett and Whitehead[1] or are implicitly targeted by the current evaluation functions[2].

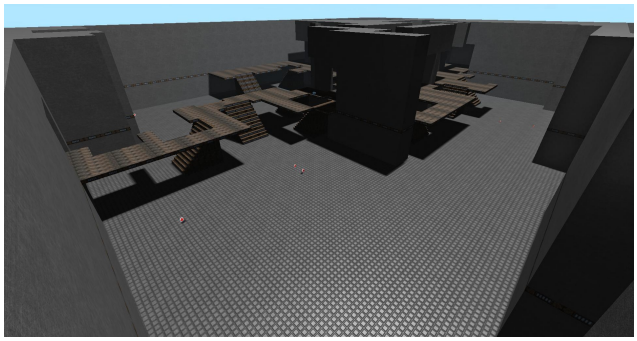


Figure 1: An evolved multi-floor shooter level in *Cube 2*.

Methodology

This section describes how shooter level topology is compactly represented as a genotype, how this genotype evolves and which criteria are used for evaluating it.

Shooter Level Representation

For the purposes of this generator, the minimal components necessary for shooter levels are considered. These components are topological (rooms, corridors and stairs) as well as game-specific (spawnpoints and powerups). Topological components form the level architecture in the *Cube 2* engine[3], using two different representations depending on the floor being created. For the first floor, corridors and rooms are placed and interconnected; corridors and rooms define this floor's *passable* tiles, and anything beyond these tiles consists of walls. For the second floor, a *random digger* approach is used: an agent converts impassable space into passable by moving and turning at random (guided by probabilities) and adds stairs to the first floor. The height of the first floor's walls extends well beyond the height of the two floors, allowing players from the second floor to jump down and join the action. The digger begins at the center of the map; when it adds a passable tile (by moving forward), a

[1]For instance, the use of a digger for the second floor to create flanking routes and hidden areas and, on the first floor, the use of rooms to form arenas and corridors to form chokepoints.

[2]For instance, the exploration fitness rewards multiple paths (which can act as flanking routes) between two spawnpoints.

[3]http://sauerbraten.org/

(free-floating) balcony is created if the floor below has passable tiles or a tunnel is dug out if the floor below has walls.

Corridors and rooms can only be placed on the first floor, while the random digger can only operate on the second floor; this creates open spaces for arenas (Hullett and Whitehead 2010) on the first floor, while the random digger is likely to create narrow, winding corridors which provide flanking routes to the arenas below, or create narrow balconies which can act as a gallery (if placed above a corridor) or a sniper location (if placed above an arena) as per the patterns of Hullett and Whitehead. Moreover, the continuous "trail" of the digger ensures that all areas in the second floor are connected, and therefore accessible to all players.

Once the architecture has been created, scripts add locations where players start the game and resurrect when they are killed (spawnpoints) and locations where powerups appear at regular intervals. Powerups can be healthpacks and weapons: a healthpack powerup increases a player's health up to a maximum value, while a weapon powerup increases a player's available shots with a specific weapon (unlocking that weapon if a player did not posses it). Gameplay follows standard tropes of the genre: players start the game at a spawnpoint (chosen randomly) and need to find and kill their opponents while also managing their health by collecting healthpacks and ammunition for their more powerful weapons (rifles in this study). Upon dying, players reappear at full health at a randomly chosen spawnpoint.

The components of the shooter level (the phenotype) are represented in the genotype as an array of real numbers. Rooms are represented by their central coordinates and size, corridors are represented by their coordinates and the corridor's width and length (negative length aligns corridors vertically rather than horizontally). The second floor's digger is represented as five probability values: the probability to move forward, to turn left, to turn right, to move onto a visited cell, to place a flight of stairs. To avoid a low locality in the search space by having a digger create different second floor "trails" from one generation to the next, a seed for all diggers' randomness is set at the start of evolution. Since spawnpoints and powerups can be placed on either floor, a spawnpoint is represented by its Cartesian coordinates and its floor; in addition to those values, powerups include a type value, which determines if the powerup is a healthpack or a weapon pickup. To sidestep the imbalance between weapons, all weapon powerups in the reported experiments are rifles.

Shooter Level Evaluation

Evaluating shooter levels is not straightforward: the current generator evaluates levels based on simulations with artificial agents, based on the available space for navigation and based on the presence of specific level patterns.

Simulation-based metrics are derived from brief playthroughs of four artificial agents, which are included in the *Cube 2* software and are set on an average skill level (simulating a more human-like aiming skill). The simulations set out to test how well the level accomplishes two design goals: (a) have players engage their opponents as much as possible and (b) maintain a balance in player kills. Goal (a) emulates the frantic pace of an FPS deathmatch, and penalizes long periods of downtime where players search for their opponents when the game starts; however, it also penalizes instances where opponents kill each other quickly due to the lack of cover, escape routes, or healthpacks (all of which prolong the fight). This goal is evaluated based on the artificial agents' *fighting time*, initially used by Cardamone et al. (2011), which is calculated from the moment a player engages any opponent to the moment the player is killed. Total fighting time ($\bar{T}_f$) is averaged from all players in the game and all their "lives" (i.e. the times they respawned); this total fighting time is used to drive evolution. Goal (b) aims to ensure that all players have an equal chance of winning, and is in line with the player balance game design pattern (Björk and Holopainen 2004). An equal kill count among artificial agents (of the same skill level) means that there is little impact from the initial spawn locations of players. The *kill ratio* heuristic (K_r), initially used by Lanzi, Loiacono, and Stucchi (2014), was extended for more than two opponents (four in this study) and is calculated as per eq. (1).

$$K_r = -\sum_{i=1}^{B} \frac{K_i}{K} \log_B \left(\frac{K_i}{K}\right) \quad (1)$$

where B is the number of artificial agents in the simulation; K is the total number of opponents killed by all agents; K_i is the number of opponents which were killed by agent i.

Another heuristic introduced by Cardamone et al. (2011) is *free space*. Levels with enough free space are easier to navigate than those with narrow corridors, although the latter allow players to take cover and hide, which can increase fighting time. The free space metric (S) is calculated as the number of passable tiles on both floors, divided by the map's dimensions. Note that for the second floor, S ignores areas which are open-air (allowing players to jump) but cannot be traversed via e.g. ramps or balconies.

In order to specifically adjust the locations of powerups and spawnpoints, this paper uses the generic methods for evaluating game levels of Liapis, Yannakakis, and Togelius (2013), adapting them to the needs of shooter levels. The design goals for object placement are: (a) to place spawnpoints faraway from each other and in difficult to discover locations (to avoid camping behaviors) and (b) to have some powerups nearby where players spawn but also some powerups far away to prompt level traversal. Goal (a) is achieved with *area control* (f_a) and *exploration* (f_e) heuristics and their balance dimensions (b_a, b_e) in eq. (2-5). Goal (b) is achieved with *powerup distribution* (f_p) in eq. (6); f_p rewards levels where an equal amount of powerups is in safe areas (near spawnpoints) as in unsafe areas (far from all spawnpoints).

$$f_a = \frac{1}{W} \sum_{i=1}^{W} A_i \quad (2)$$

$$b_a = 1 - \frac{1}{W(W-1)} \sum_{i=1}^{W} \sum_{\substack{j=1 \\ j \neq i}}^{W} \frac{|A_i - A_j|}{max\{A_i, A_j\}} \quad (3)$$

$$f_e = \frac{1}{W} \sum_{i=1}^{W} E_i \quad (4)$$

$$b_e = 1 - \frac{1}{W(W-1)} \sum_{i=1}^{W} \sum_{\substack{j=1 \\ j \neq i}}^{W} \frac{|E_i - E_j|}{max\{E_i, E_j\}} \quad (5)$$

$$f_p = 1 - |2\frac{P_s}{P} - 1| \quad (6)$$

where W and P is the number of spawnpoints and powerups, respectively; A_i is the area of passable tiles which are *safe* for spawnpoint i; E_i is the exploration effort (simulated as flood fill area coverage) to reach all other spawnpoints starting from spawnpoint i; P_s is the number of powerups in a safe area (for any spawnpoint). *Safety* of a tile to a spawnpoint is measured as the distance from that spawnpoint to the closest other spawnpoint; if the safety score is above a constant value (0.35 in this study), the tile is considered *safe*.

Shooter Level Evolution

In order to more efficiently search the large space of possible levels, the generator described in this paper evolves shooter levels in two stages. Each stage evolves different parts of the genotype, and attempts to limit the breadth of fitness dimensions being optimized. The first stage evolves the architecture of the level, placing rooms and corridors and finding the right probabilities for the second floor's digger. The second stage evolves the placement of game objects (powerups, spawnpoints) on an otherwise static level. The rationale for the two-stage process (and the order of its stages) is that architecture needs to be finalized before the finer details such as powerups can be placed. The fittest individual of the first stage defines the architecture of the shooter level used in the second stage; all individuals in the second stage use the same architecture, changing only the position and type of powerups and spawnpoints.

Since the different stages in the proposed process evolve different aspects of the shooter level, the fitness function for each stage needs to be carefully designed. Levels are evolved to maximize F_{arc} in eq. (7) in the first stage: the first stage evaluates available space for gameplay (S) and attempts to secure an adequate fighting time and kill ratio for the artificial agents in the simulations. The second stage places more emphasis on object placement evaluated via F_{obj} in eq. (8); however, simulation-based heuristics are retained since simulations can capture the emergent impact of the moving game objects. Evolution in the second stage attempts to optimize $F_{arc} + F_{obj}$.

$$F_{arc} = \tfrac{1}{3}(\bar{T}_f + K_r + S) \quad (7)$$

$$F_{obj} = \tfrac{1}{5}(f_a + b_a + f_e + b_e + f_p) \quad (8)$$

The first stage evolves levels' architecture disregarding spawnpoints and powerups when doing so; however, spawnpoints are essential for gameplay (otherwise players can not start the game somewhere) while powerups play a significant role both for gameplay purposes (without rifle pickups in the level, the default weapon is very weak) and as nodes for the artificial agents' pathfinding. Rather than bias search by randomly placing powerups and spawnpoints in the evolving levels or by randomly mutating object placement while architecture also changes (as the object is likely moved inside a wall), levels in the first stage of the two-stage evolution use the method of Cardamone et al. (2011) for uniformly distributing objects in the level. This method is constructive, and divides each floor into 16 blocks of equal size: if sufficient free space exists within the block, one powerup and two spawnpoints are placed inside the block.

In both stages, levels are evolved via mutation alone. During evolution, parents to mutate are chosen based on a tournament selection of size 3, i.e. three individuals are selected from the previous population at random and the fittest among them becomes a parent. Every level component described in the chosen parents' genotype has a chance of mutating; most level components have a 10% probability of mutating, although certain probabilities were adjusted based on preliminary testing. Mutation adjusts the current parameters, increasing or decreasing them by a random value; for instance, coordinate mutation of a powerup moves it up or down and left or right from its current position while type mutation of a powerup may change it from weapon to healthpack. Due to the admittedly high mutation rates, repair functions for each stage of evolution correct any errors introduced during mutation. In the first stage, if mutation renders a room or corridor inaccessible from the rest of the level, then the repair function removes it from the phenotype (replacing it with walls). In the second stage, if mutation moves a powerup or spawnpoint outside passable areas, then the repair function moves it to a random passable area on the same floor. The entire population is replaced by offspring of fit parents, except for the three fittest individuals which are transferred to the next generation (i.e. elitism of 3).

It should be noted that in the first stage of evolution, only the genetic information for rooms, corridors and digger probabilities is used to create the level, with powerups and spawnpoints placed in a constructive fashion described above. In the second stage of evolution, only the coordinates and types of powerups and spawnpoints are used to create the level, with the architecture loaded from the final best individual of the first stage. This reduces the number of genetic parameters which need to be optimized per stage.

Experiments

This paper evaluates the generator based on its behavior during evolution and — primarily — based on the shooter levels it produces. The two-stage evolutionary method proposed is used to generate shooter levels with two floors, as well as shooter levels with one floor (by omitting the random digger). Results are collected from 10 independent evolutionary runs for each type of level. Each run optimizes a population of 50 individuals for a total of 60 generations; the first stage evolves towards F_{arc} of eq. (7) for 30 generations and the second stage for $F_{arc} + F_{obj}$ of eq. (8) for another 30 generations. To control for the random nature of deathmatch games, simulation-based evaluations are averaged from five 2-minute simulations with four artificial agents.

Optimization Behavior

The final results of two-stage evolutionary runs for shooter levels with one or two floors are collected in Table 1. The values are averaged from 10 independent runs of each method, with standard deviation included in parentheses. It is clear that levels with a second floor, when optimized, can achieve a higher $\bar{T}_f$ metric than levels with a single floor; comparisons between two-floor and one-floor level evolution shows significant differences using a two-tailed Mann-

	Two floors		One floor	
Fitn.	Fittest	Average	Fittest	Average
T_f	30.0 (4.8)	22.0 (1.8)	23.2 (3.8)	17.7 (1.9)
K_r	0.90 (0.11)	0.84 (0.04)	0.96 (0.03)	0.86 (0.04)
S	0.76 (0.08)	0.76 (0.05)	0.62 (0.05)	0.60 (0.03)
F_{arc}	0.89 (0.11)	0.78 (0.06)	0.88 (0.11)	0.69 (0.05)
f_a	0.46 (0.05)	0.43 (0.05)	0.48 (0.06)	0.54 (0.07)
f_e	0.54 (0.05)	0.52 (0.04)	0.49 (0.06)	0.52 (0.04)
b_a	0.67 (0.09)	0.62 (0.07)	0.60 (0.10)	0.67 (0.08)
b_e	0.95 (0.02)	0.93 (0.01)	0.91 (0.03)	0.92 (0.01)
f_p	0.90 (0.05)	0.87 (0.02)	0.89 (0.12)	0.89 (0.03)
F_{obj}	0.70 (0.01)	0.67 (0.01)	0.67 (0.04)	0.71 (0.01)

Table 1: Scores of the final evolved levels at the end of the 2nd stage, both for the overall fittest levels (with regards to $F_{arc} + F_{obj}$) and the average in the population. Note that $\bar{T}_f$ is in seconds of fighting time; when calculating F_{arc} this value is divided by 30.

Whitney U-Test ($p < 0.01$) for both the maximum and average values of $\bar{T}_f$. How the introduction of an additional floor affects gameplay and agent behavior will be elaborated, with examples, in the next section. The S metric is unsurprisingly significantly higher for two-floor levels, since the second floor offers additional free space for navigation. However, remaining scores of Table 1 do not differ between one-floor and two-floor evolution. Another interesting finding is that K_r is high for both types of levels; an obvious reason is that the competing artificial agents have the same skill level and decision making processes. Another likely reason is the fact that on death, players spawn at a random spawnpoint; with enough respawns, all players start from all spawnpoints and therefore have access to the same affordances (health, weapon pickups). Contrary to K_r, most of the evaluations pertaining to game object placement do not reach high values (as demonstrated by F_{obj}); it is unclear what prompts this behavior, as F_{obj} has an equal impact to evolution as F_{arc} in the second stage. It could be presumed that the mutation scheme which changes spawnpoint and powerup coordinates either leads to small changes which do not improve F_{obj} substantially in 30 generations, or leads to destructive changes which cause the repair function to often place spawnpoints and powerups in random new locations (reducing the locality of search). Another reason for this behavior is due to the fact that the first stage has already optimized F_{arc}; high values in F_{arc} as a result of the first stage can dominate the sum of fitnesses and lead to sub-par optimization of F_{obj}. This is obvious for one-floor levels in Table 1, since the fittest levels in $F_{arc} + F_{obj}$ have a lower F_{obj} score than the population's average.

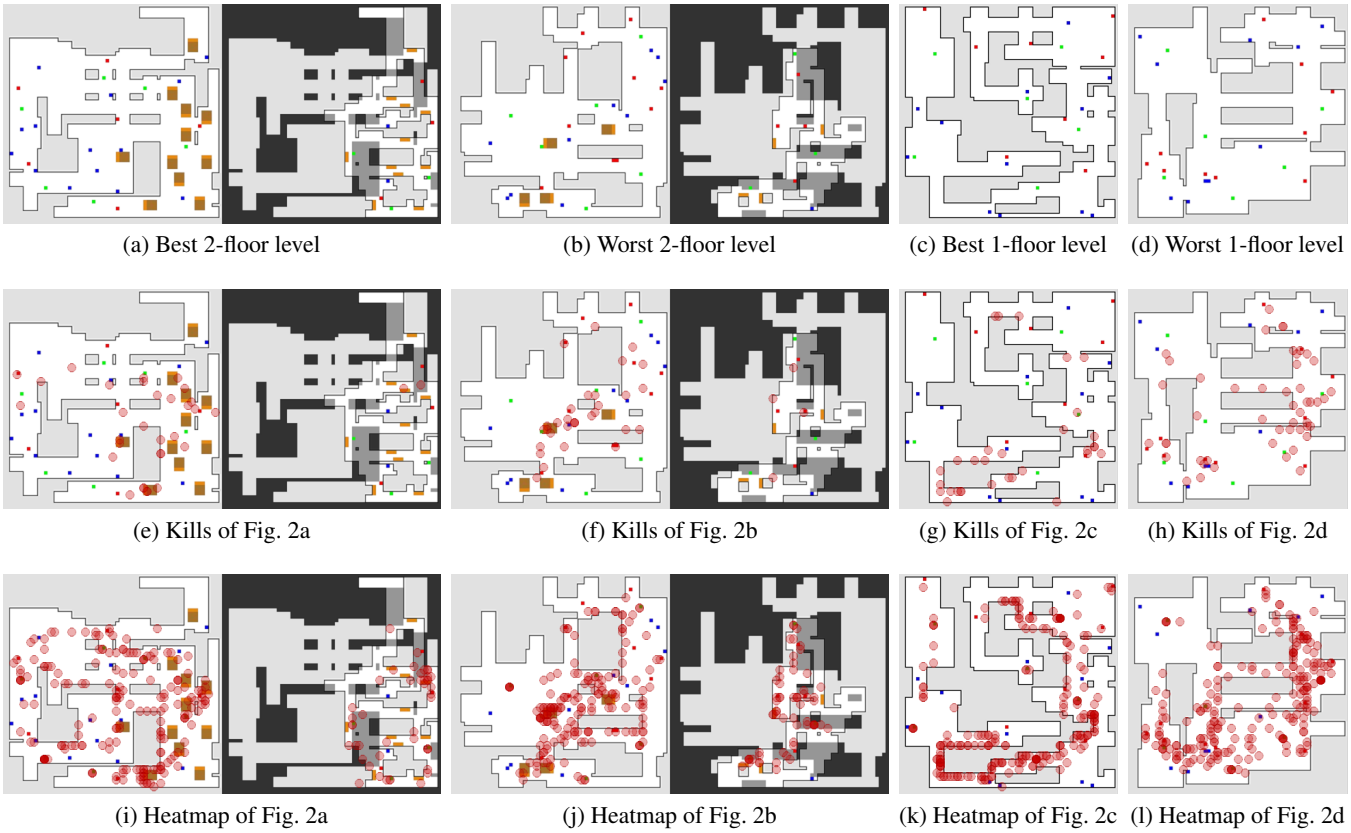
Sample generated levels and their patterns

In order to better understand how the heuristics' scores of Table 1 actually translate into shooter levels, it is worthwhile to investigate the behavior of the artificial agents evaluating them. Indicatively, this paper will show the fittest among the best evolved levels of the 10 independent runs and the least fit among them; this should demonstrate how different level patterns can affect both its fitness score and agents' behavior. Figure 2 shows the best and worst among the fittest levels at the end of the second stage of evolution; Figure 1 also shows the level of Fig. 2a within the *Cube 2* environment. While the first floors of all four levels superficially look similar, with large rooms and winding corridors, there are obvious differences in the second floor setup between Fig. 2a and 2b. The random digger of Fig. 2a seems to have a high probability of adding stairs, as 11 stairs connect the floors (compared to 4 stairs in Fig. 2b). Regarding object placement, both Fig. 2b and Fig. 2d have spawnpoints placed relatively close to each other, in the same central arena formed by multiple rooms and corridors, while Fig. 2a better disperses spawnpoints in the level. When a single floor is evolved, maps seem "busier" than when two-floor levels are evolved, i.e. have more winding maze-like corridors and dead-ends, especially in Fig. 2c. A likely explanation for this level structure is that, without a second floor with tunnel (dug-out) sections to provide cover, the single floor must rely on winding corridors with healthpacks to provide cover and confuse opponents, thus ensuring player survival and thus increase fighting time.

Figures 2e-2l show death locations and heatmaps of the simulations used to evaluate these maps. It seems that most agents die on the first floor regardless of the architecture and accessibility of the second floor in Fig. 2a and 2b. This is a persistent pattern in all 10 of the final best evolved two-floor levels: only 16% of agent deaths occur on the second floor. Interestingly, the numerous stairs to the second floor in Fig. 2a do not result in more visits of agents to the second floor as shown in Fig. 2i: 19% of the agents' heatmaps for Fig. 2a are on the second floor, compared to 27% for Fig. 2b. Another interesting pattern is that the top-most flight of stairs in Fig. 2i is never visited, despite a convenient shortcut via the second floor; there is little incentive to visit that area of the first floor as it contains a single healthpack. For one-floor levels, the winding corridors and smaller arenas of Fig. 2c result in movement and combat (and deaths) to be concentrated at the narrow corridors of the bottom of the map. It should be noted that level architecture is not the only factor in this behavior, as the narrow corridors at the bottom of Fig. 2c divide the level into two otherwise disjointed map halves, with two spawnpoints on the left half and four on the right half. Comparatively, the architecture of Fig. 2d has more pathways between different parts of the map, leading to a more evenly distributed agent heatmap.

Discussion

Results of evolution highlighted several contributions to the procedural content generation of shooter levels, as well as several limitations that should be addressed in future work. The evolution of levels in a two-stage process, first with evolving architecture and then with evolving game object placement, resulted in levels which allowed players to engage in combat for longer periods of time — although object placement heuristics seem to be dominated by the first stage's fitnesses. A more interesting finding is the impact that an additional floor has on fighting time, as it allows for players to "timeout" from combat (which predominantly



(a) Best 2-floor level (b) Worst 2-floor level (c) Best 1-floor level (d) Worst 1-floor level

(e) Kills of Fig. 2a (f) Kills of Fig. 2b (g) Kills of Fig. 2c (h) Kills of Fig. 2d

(i) Heatmap of Fig. 2a (j) Heatmap of Fig. 2b (k) Heatmap of Fig. 2c (l) Heatmap of Fig. 2d

Figure 2: Best and worst among the fittest evolved shooter levels among 10 individual runs. For two-floor levels, the left map is the first floor while the right map is the second (top) floor. Large brown rectangles represent stairs to the second floor, red tiles represent weapons, blue tiles represent healthpacks and green tiles represent spawnpoints. On the second floor map, white areas are free-floating balconies above the first floor, while dark gray areas are dug out from the walls of the first floor (and signify tunnels); light gray shapes represent the floor below it, to show where players can jump off to or aim at from the second floor.

happens on the first floor, based on heatmaps of several generated levels) and explore otherwise less visited locations.

An important caveat in the findings of this paper is the assumption that the artificial agents used during simulations accurately reflect how human players would perform in the same levels. If the behavior of agents is not as human-like as would be desired, this can affect the evaluations of quality which drive evolution (i.e. fighting time and kill ratio) but also the conclusions drawn from playtraces of the sample generated levels. For instance, while artificial agents may travel only towards powerups such as healthpacks and weapons, human players may seek out-of-the-way areas (devoid of powerups) to hide from combat, or to stage an ambush. Future work must validate this paper's findings with human players competing against each other in a few of the best generated levels of this method.

Optimizing the many different heuristics of shooter levels by aggregating them in a weighted sum was expected to lead to sub-par performance of the evolutionary algorithm. The use of a two-stage process where the first stage only optimizes a subset of the heuristics was expected to mitigate this problem, although this paper does not include comparisons with simpler approaches (search-based or constructive) for the sake of brevity. A performance boost could also be achieved by limiting the number of heuristics by testing which ones result in better levels[4] and are not conflicting with vital metrics such as fighting time. Other solutions include introducing more stages of evolution with fewer moving parts and simpler evaluations, or using multi-objective evolutionary methods (Coello 1999).

While future work should focus on improving the quality of evaluations (and their fidelity with human play) as well as the genetic algorithm, another important area for improvement is the presentation of the final levels. This is of paramount importance if the generated levels are intended for human play (e.g. as part of a verification process of current findings), since bland and repetitive rooms can quickly lead to user fatigue. Procedurally generating a more elaborate presentation of the final levels can use constructive approaches and include placing different wall and floor tex-

[4]For instance, the high values of K_r for all generated levels indicates that it could be omitted as a fitness dimension since players respawning at random locations evens out most map imbalances.

tures in different rooms, making bridges and balconies more obvious (with e.g. railings, curved arches, or buttresses) and adding light sources or sound sources to help navigation.

Conclusion

This paper presented a method for representing and evolving levels for first person shooter games which span more than one floors. Several heuristics were proposed for evaluating these levels, which were inspired from earlier work on simulation-based FPS level generation and theory-driven evaluations of typical level design patterns. The heuristics primarily targeted competitive play in deathmatch-style game setups popular in the shooter genre. Finally, the paper proposed and tested a two-stage method for generating levels by initially evolving their architecture and subsequently the placement of game objects within them. Experiments showed that a second floor affects both the simulated players' movement patterns and the time they spend in combat. Moreover, different level patterns emerged in the generated levels, and their effect on gameplay was observed.

Acknowledgments

The research was supported, in part, by the FP7 ICT projects C2Learn (project no: 318480) and ILearnRW (project no: 318803), and by the FP7 Marie Curie CIG project AutoGameDesign (project no: 630665).

References

Alexander, C.; Ishikawa, S.; and Silverstein, M. 1977. *A Pattern Language*. Oxford University Press.

Björk, S., and Holopainen, J. 2004. *Patterns in Game Design*. Charles River Media.

Cardamone, L.; Yannakakis, G. N.; Togelius, J.; and Lanzi, P. L. 2011. Evolving interesting maps for a first person shooter. In *Proceedings of the Applications of evolutionary computation*, 63–72. Springer-Verlag.

Coello, C. A. C. 1999. A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information Systems* 1(3):129–156.

Cook, M., and Colton, S. 2011. Multi-faceted evolution of simple arcade games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*.

Hullett, K., and Whitehead, J. 2010. Design patterns in fps levels. In *Proceedings of the 5th Conference on the Foundations of Digital Games*.

Lanzi, P. L.; Loiacono, D.; and Stucchi, R. 2014. Evolving maps for match balancing in first person shooters. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*.

Liapis, A.; Smith, G.; and Shaker, N. 2015. Mixed-initiative content creation. In Shaker, N.; Togelius, J.; and Nelson, M. J., eds., *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.

Liapis, A.; Yannakakis, G. N.; and Togelius, J. 2013. Towards a generic method of evaluating game levels. In *Proceedings of the AAAI Artificial Intelligence for Interactive Digital Entertainment Conference*.

Ølsted, P. T.; Ma, B.; and Risi, S. 2015. Interactive evolution of levels for a competitive multiplayer fps. In *Proceedings of the IEEE Congress on Evolutionary Computation*.

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

Togelius, J.; Preuss, M.; Beume, N.; Wessing, S.; Hagelbäck, J.; Yannakakis, G. N.; and Grappiolo, C. 2013. Controllable procedural map generation via multiobjective evolution. *Genetic Programming and Evolvable Machines* 14(2):245–277.

Togelius, J.; Nelson, M. J.; and Liapis, A. 2014. Characteristics of generatable games. In *Proceedings of the FDG Workshop on Procedural Content Generation*.

Uriarte, A., and Ontañon, S. 2013. PSMAGE: Balanced map generation for starcraft. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*.

Yannakakis, G. N., and Togelius, J. 2011. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing* 2(3):147 – 161.