This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2022.3192930

Genetic-WFC: Extending Wave Function Collapse with Genetic Search

Raphael Bailly, Guillaume Levieux

***Abstract*—This paper presents Genetic-WFC, a procedural level generation algorithm that mixes genetic optimization with Wave Function Collapse, a local adjacency constraints propagation algorithm. We use a synthetic player to evaluate the novelty, safety and complexity of the generated levels. Novelty is maximized when the synthetic player goes on tiles not visited for a long time, safety is related to how far it can see, and complexity evaluates the variability of the surrounding tiles. WFC extracts constraints from example levels, and allows us to perform the genetic search on levels with few local asset placement errors, while using as little level design rules as possible. We show that we are able to rely on WFC while optimizing the results, first by influencing WFC asset selection and then by re-encoding the chosen modules back to our genotype, in order to optimize crossover. We compare the fitness curves and best maps of our method with other approaches. We then visually explore the kind of levels we are able to generate by sampling different values of safety and complexity, giving a glimpse of the variability that our approach is able to reach.**

***Index Terms*—Procedural Content Generation, Level Design, Player Experience, Variability, Wave Function Collapse, Genetic Algorithm, Video Games**

I. Introduction

GAME levels procedural generation can be approached with constructive as well as generated-and-test algorithms [1]. Constructive algorithms mainly rely on design knowledge to create levels in one go, without the need to further evaluate and refine them. Generate-and-test approaches mainly focus on optimization algorithms that search the game-level space, iteratively creating levels to maximize a specific fitness function. The goal of this paper is to propose a level generation pipeline that mixes a search-based algorithm with a generic constructive one, and show how these two methods benefit from each other. More specifically, we show that a generic, data-based constructive algorithm can help a search-based algorithm to deal with a constrained optimization problem without fitness penalization. The constructive algorithm takes care of the constraints, while the search-based algorithm can focus on improving the simulated game experience.

Constructive and generate-and-test approaches are complementary. The design knowledge on which constructive methods rely can allow to quickly build levels of high quality. However, if a game level can be simulated and evaluated, one may further refine the levels to make them even better, tune the game experience they provide, or allow for more flexibility and randomness as bad levels should be detected and discarded. Search-based methods explore a wide generation space, as they are only limited by the fitness score they obtain when modifying the levels. However, searching the levels space can be a time-consuming process, as many bad or unfeasible levels will be generated and tested. We thus propose to use a generic, data-driven constructive method known as Wave Function Collapse (WFC), that encodes basic knowledge about the layout of the level, to boost a genetic optimization algorithm (GA). Indeed, using WFC as a repair operator, the GA will only manipulate levels exempt of basic asset placement errors and concentrate on improving the simulated game experience.

Wave Function Collapse seems particularly adapted to correct the errors of a search-based level generator. WFC has shown its capacity to generate infinite virtual cities [2], and has been used in commercial games and tools [3], [4]. This algorithm extracts adjacency constraints from example levels, and then applies them to generate new levels of arbitrary size. This algorithm is not limited to a specific game genre, as it can be applied to any grid-based procedural level generator that needs to enforce adjacency constraints.

To better evaluate this Genetic-WFC algorithm, we focus on generating semi-opened levels, that do not constrain the player along a unique, restricted path, but allow them to move more freely in their environment. In our experiment, the player has a specific location to reach in the level. The level contains blocking geometry, both for player vision and navigation, and enemies navigate the level to attack the player and prevent them from reaching their goal too easily. For instance, *Ghost Recon* or *Far Cry* enemy camps typically offer this kind of experience for the shooter genre, but so does an *Elden Ring*'s dungeon, to only name a few [5]–[7].

Such games provide complex environments as well as an emergent gameplay. They are thus hard to generate for both constructive and generate-and-test approaches, and will provide an interesting test bed for our algorithm. For instance, when building an enemy camp, stairs can't be placed in the middle of the road, and must lead to a valid position. Fences should be around the camp and not inside, and vehicles can't be parked anywhere. These rules constrain the relative placement of assets, and they might be used to design a constructive algorithm that could generate a valid camp. However, assets placement will also block the path and visibility of both the player and their enemies as they freely roam the level, and the game experience provided by the level will emerge from this setup. Building a constructive algorithm for emergent levels is much harder, as simply adding a crate on top of another

Raphael Bailly and Guillaume Levieux were with the CEDRIC lab, Conservatoire National des Arts et Métiers (CNAM), Paris, France e-mail: (guillaume.levieux@cnam.fr).

This work was funded with the help of Nouvelle-Aquitaine, for the Kiwi Project, under grant agreement No 18001957, and with the help of BPI with the United VR project.

one may block the visibility of a sniper patrolling nearby, and create an opportunity for the player to move safely toward their goal, changing the whole game experience. For this aspect of the gameplay, it might be interesting to create a search-based algorithm that simulates the game, rates the levels accordingly and iterates towards the best solution. This gameplay will thus allow us to test Genetic-WFC generation capabilities, and it is generic enough to show how our algorithm may perform on many games that share these design principles.

In this paper, we thus propose a search-based level generation pipeline that uses WFC as a repair operator to generate semi-open FPS-like levels. We describe our solution and how we combine a genetic algorithm with WFC. Then, we evaluate WFC in terms of computation cost, as the main drawback from this repair mechanism is the additional amount of computation. We then compare our approach with a pure genetic algorithm and a penalized genetic algorithm. Lastly, we will visually explore the levels that Genetic-WFC can generate for different values of complexity and safety.

II. Related Works

A. Wave Function Collapse

Wave Function Collapse (WFC), is a procedural generation algorithm based on constraint propagation, initially introduced in order to reproduce textures [8]. It has since been used in different domains, including game level generation. The game *Bad North* uses WFC to create 3D islands, and the generator was then turned into a mixed-initiative generator called *Townscaper* [3], [4]. Other games also implement single [9], [10] or multi-layered WFC [11].

The WFC algorithm is based on a grid. Each cell of the grid can contain a module [8]. A module is linked to a graphical asset, and must respect adjacency constraints: it contains a list of the other modules that can be placed next to it, for a given rotation around the up axis. Initially, each cell may contain any module, with equal probability. At each step, a random cell among the cells with the smallest number of available modules will randomly select one module, following the probability distribution among available choices. These cells have the lowest *entropy*, as they are the most constrained cells (see Sec. III-E and Eq. 1). When this choice is made, the number of possibilities available for the neighboring cells is reduced, because some modules can't be placed anymore, as they would violate adjacency constraints. Removing these available modules will again trigger updates for the neighboring cells, until no more updates are required. Then a new random choice is made.

WFC can be applied using two different approaches: the Overlap Model and the Simple Tiled Model. The Overlap Model uses an initial grid and divides it into overlapping subregions, to drive generation with a list of allowed patterns. The Simple Tile Model will only use a list of adjacency constraints, either manually set or extracted from an example grid. WFC can be either used in two or three dimensions. The shape of the grid may also vary, as well as the number of neighboring cells taken into account [12].

This algorithm seems promising to assist a search-based algorithm for level generation. It has already been used to generate playable levels by only propagating simple local constraints, and may thus allow us to focus on evaluating player experience for levels without basic asset placement errors. This algorithm has already been extended to add more complex level design constraints [13], and we follow a similar approach by modifying the selection probabilities of the modules. This way, we can use an iterative search algorithm on top of WFC, as described hereafter.

B. Iterative Search

Game levels can be generated by computing multiple iterations of the generation stage until a good enough solution, according to a certain fitness function, is reached. A way to perform this iterative search is to rely on an evolutionary algorithm to optimize, step by step, a population of candidate levels [14]. This approach has for instance already been applied in a 2D platform game [15], as well as for the creation of playable maps for a real-time strategy game [16].

Iterative search algorithms need to constantly evaluate the levels they produce. This evaluation step can be performed by a synthetic player. Several studies have focused on the use of autonomous agents called personas, with different goals, in order to evaluate and test the playability of dungeon levels, for example [17], [18]. We follow the same approach in this work, as we will drive the generation by evaluating a level from a synthetic player's point of view. Indeed, as we explained before, the levels we want to generate provide an emergent gameplay, and a single local change may modify the resulting experience, by blocking a path, opening a shortcut or providing a cover for instance. By simulating the gameplay, we should be able to evaluate the emerging game experience, given our current synthetic player's persona.

However, it should be noted that the major disadvantage of iterative search and synthetic player simulation is the amount of computation necessary to explore and simulate the game level space. As a result, gameplay simulation and level generation must be kept as simple as possible to ensure that the level-space exploration is achievable.

From a Genetic Algorithm point of view, the problem we are trying to solve can be considered as a constrained optimization problem. Our levels both require to provide and certain play experience, as well as respect relative asset placement constraints. There exist multiple ways to perform a constrained optimization with a GA [19]. First, one may penalize the fitness function of the search algorithm every time a constraint is violated. However, such a penalty might be hard to design. Placing a stair in the middle of the road is a big mistake, but if we penalize it too much, then we might lose the good features of the rest of the level as it will be destroyed. In order to avoid balancing one function between the simulation score and constraints satisfaction, another possibility is to place levels that do not respect constraints in a different population and evolve them differently, only to correct the constraint-related errors. FI-2Pop is such an algorithm. It maintains two populations of individuals, the feasible and infeasible, and any individual can move from one population to another, if it either respects or not the constraints, and evolve to either maximize a fitness function or the amount of broken constraints [19], [20]. Finally, another way is to use specific steps in the algorithm to correct the errors of a given solution, that can be called a repair operator. However, repair operators are often not generic solutions, and need to be specifically designed for each optimization problem, and can be costly in terms of computation time when repairing individuals.

In this research, we are focused on game-level generation, and WFC is generic enough to our domain to be used as a repair operator. By only generating levels with WFC, we should limit our search space to levels that respect our simple constraints. As a first step, we thus need to describe and evaluate Genetic-WFC, showing if indeed WFC can be considered a valid candidate as a repair operator. However, in further work, it would be very interesting to place ourselves in the full spectrum of constrained genetic algorithm and explore how our algorithm may compete with algorithms like FI-2Pop.

Given these previous works in procedural generation of game levels, it seems that a mixed approach, that uses both an iterative search and WFC might be promising. It will allow us to avoid the evaluation of levels that do not respect simple level design constraints and to focus our iterative search on gameplay simulation and scoring. To be more specific, for instance, WFC will enforce the simple rule that stairs can't be placed if they lead to nowhere or to a blocking wall. This kind of error will not be evaluated by the synthetic player, that will be focused on the global gaming experience, and will not have as much semantic information about the asset as we do. As far as the synthetic player is concerned, stairs are just another navigable tile, and climbing a set of stairs to then just jump into the air is a way to explore the level. WFC may allow us to enforce a correct level structure, extracted from example levels, to prevent this kind of obvious design error. However, such an approach needs to be evaluated in terms of computation cost as well as its capability to generate a varied set of interesting levels, which is what the reminder of this paper is focused on.

III. Genetic-WFC

A. Algorithm Overview

Genetic-WFC is a generation pipeline based on a genetic algorithm, that uses WFC to only generate candidate levels that respect specific asset placement constraints, as well as to transform a greyblock level into a final one. Figure 1 describes this pipeline, and we detail it hereafter.

As can be seen at the top of Figure 1, our genetic algorithm drives the WFC to generate multiple levels, using *boost zones* to influence asset selection probability, as described in section III-E. This WFC extracts the constraints by only validating the relative asset placement present in example grids. Also, as we explain in section III-D, we only manipulate greyblock assets at this stage of our pipeline. WFC computing time grows with the number of modules, and many graphical assets have the same level design function. So we first generate the level using a restricted number of modules, our greyblocks.

Then, the resulting greyblock grid is annotated with navigability and visibility information, later used by the synthetic player, detailed in section III-F, during the simulation step. Our agent will rate the level, depending on its current persona, i.e. the current weightings for the novelty, safety and complexity ratings.

These previous steps are performed iteratively on a population of candidate levels, as long as we do not meet a specific termination criteria. For our experiment, we simply perform a fixed number of loops.

Then, the greyblock levels with the best fitness is kept, and is processed by another WFC. Each greyblock level corresponds to a category of assets, e.g. stairs or fences. As explained in section III-D we use these categories and the constraints relative to the graphical assets to generate the final level, a level with graphical assets corresponding to the greyblock level. This step can be run multiple times to generate levels with the same navigation and visibility properties, but with different graphical assets.

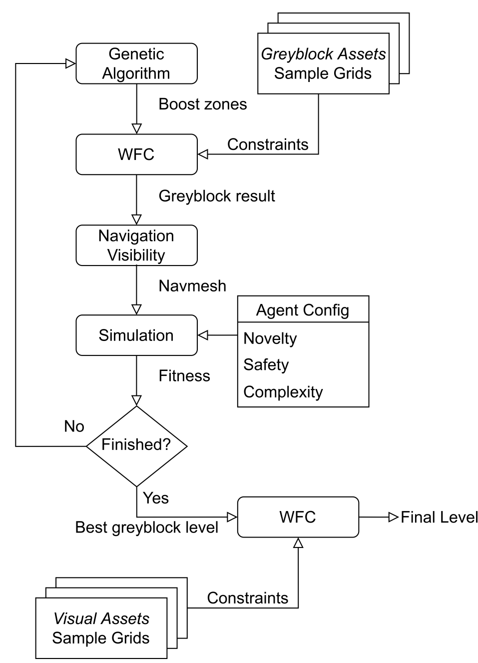


Figure 1: Evolutionary WFC Algorithm

B. Wave Function Collapse

We have implemented the Simple Tiled Model, as it is the simplest and fastest algorithm. The Overlap Model is very powerful as it reproduces patterns from the example, but we want to limit the constraints applied at the generation stage, and we need to save as much computation time as possible. Indeed, WFC will have to generate many new levels at each step of our evolutionary algorithm, which can really increase computation time. Our constraints are automatically extracted from example game levels created using WFC modules manually arranged on a 2D grid (Fig. 2 or Fig. 4a). We only take into account four neighbors: left, right, top and bottom, and each module can be placed with four 90 degrees rotations around the up axis. It should also be noted that we take into account the frequency of assets in the example grid when we randomly choose a module in a cell. The more an asset has been placed in the example grid, the higher the chances are that it will be selected. We can then run the WFC algorithm on a grid of any size to generate a level that respects the extracted adjacency constraints.

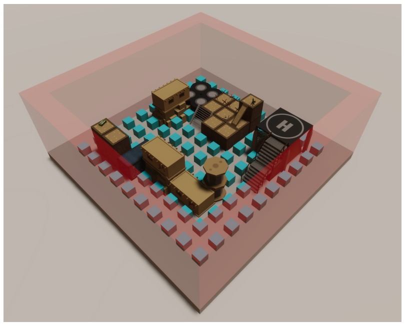


Figure 2: WFC Sample Grid, with air and border tiles.

C. Air Filling and Borders

In order to extract adjacency constraints from a manually designed level, we use two special modules that are placed automatically (Fig. 2 or Fig. 4a). First, we spawn an *air* module in every empty cell of the grid to explicitly create a link between void cells and modules. It is to note that we might also use different types of air modules to create constraints between modules that are not next to each other. We also use this trick for the border of the map, where we put a *border* asset. The edge of the map has a specific meaning, we may want to place a wall or allow circulation around the map for instance.

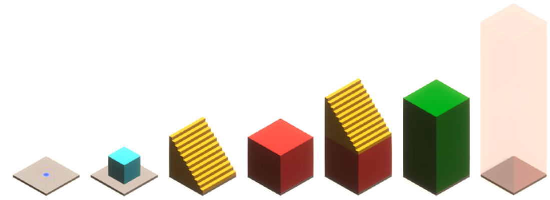
D. Greyblocking

As the number of assets grows, so does WFC computing time. However, many graphical assets will have the same gameplay function in the level. We thus decided to generate levels in two steps, and to rely on *greyblocking*, as depicted in Figure 1. We thus use a small number of modules that represent each asset categories, for instance in our examples, as shown in Figure 3: low blocking walkable volume (d), high blocking walkable volume (f), transition from floor to low walkable (c) and from low to high walkable (d). This allows us to save computation time during the iterative search, and also to mimic a working method of prototyping used by level designers: designing a level and making it a beautiful environment can be considered two separate steps. We thus add a specific step when we initialize the second-pass WFC grid: we simply remove, for each cell, all the choices that are not in the same category as the greyblock asset already in this cell (Fig. 4). This way, we can use the result of our greyblock Genetic-WFC to constrain another WFC that generates the final map.

E. Genetic algorithm for WFC

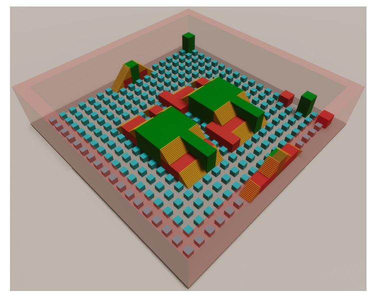
The next part of our generation system is the genetic algorithm that will drive the generation toward a specific gaming experience. In order to allow this iterative research, we need to introduce a way to control the WFC and encode it into a chromosome. To do so, we influence the selection probability of a module, every time a random choice is made by the WFC.

When the whole WFC grid is updated, the selection probability of each module in each cell is computed with regard to asset frequencies and adjacency constraints, and WFC selects

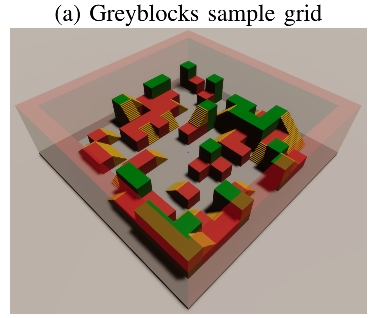


(a) Player Spawn (b) Air (c) Stairs to level one (d) Level one block (e) Stairs to level two (f) Level two block (g) Border

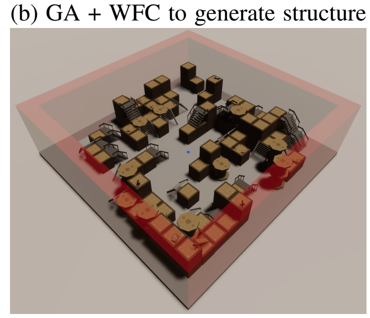
Figure 3: Modules used for greyblocking



(a) Greyblocks sample grid



(b) GA + WFC to generate structure



(c) Second WFC pass to replace greyblocks with visual assets.

Figure 4: Levels are first optimized using greyblocks to reduce complexity. Then a second WFC replaces greyblocks with visual assets.

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2022.3192930

the most constrained cell by computing each cell's entropy. For each cell that has $N > 1$ possible modules, with each module having a probability p_i to be selected, we compute entropy using equation 1. Cells where modules are equiprobable will have the highest entropy, zero. As we just need to select the lowest entropy, we do not use the original log from Shannon's entropy to save computing time.

$$H = -\sum_{i=1}^{N} \left| \frac{1}{N} - p_i \right| \tag{1}$$

a) Chromosomes Representation: In order to allow the optimization algorithm to drive the WFC, we added *boost zones* that increase the selection probability of a module in a specific region of the grid. This way, we can locally influence the asset selected by the WFC without breaking the algorithm. We only multiply the module's selection probability by a certain value for all the cells inside the boost zone. We thus can't force an asset that would violate a constraint, as its probability has already been set to zero by previous constraint propagation.

Chromosomes of our genetic algorithm thus need to encode the boost zones. In our first attempt to use boost zones, we defined a small amount of boost zones with variable sizes, variable positions, and variable probability boost factors. However, we didn't get interesting results, so we quickly switched to a more directive option with a more observable impact on the generated level. We use one boost zone per grid cell, with a fixed and very high boosting factor. As we have one boost zone per cell, we do not need to encode their size or coordinates. Our chromosome has thus the same size as the WFC grid, and only encodes the ID number of the module to boost for each cell of the grid. As a result, the genetic algorithm will choose a module for each cell of the grid, but the WFC will then translate these choices into a level that respects the adjacency constraints.

b) Chromosomes Re-encoding: To enforce a deterministic mapping of genotype to phenotype, we use the same random generator seed every time we generate a level. However, to have an efficient crossover operator, we also need to re-encode our level into its chromosome after evaluation (Algorithm 1, l.20). Indeed, our chromosomes allow us to boost the probability of a specific asset to be selected. If this asset can't be placed because it breaks adjacency rules, another one will be *randomly chosen* by the WFC. As we always use the same random generator's seed to generate a level, the same asset will always be chosen for a given cell of a given chromosome. However, this will not be the case after a crossover. WFC selects a tile with regard to the number of previously selected modules, to match the frequency of each asset in the sample grids. If an asset is not very frequent in the sample grids, and it has already been placed in the map, its selection chances will fall. Thus, when performing a crossover and mixing level together, modules wrongly placed by the chromosomes and thus chosen by the WFC alone may be different, as asset frequencies before the choice will be different in this new map. In order to prevent this loss of determinism, we simply re-encode the phenotype into the genotype when we evaluate a level. For each cell, we change the chromosome to put the ID number of the asset that has been placed in the map, as if it was the chromosome's choice in the first place. We show in section V and Figure 6 that without this re-encoding, optimizing WFC's generation is much slower, see curve *Genetic-WFC NR* for No Re-encoding.

c) Selection, Mutation and Crossover: We use the GeneticSharp library to implement our genetic algorithm [21]. The first population is initialized with random chromosomes. Parents are selected by tournament (Alg. 1, line 2). Pairs of parents are formed sequentially (Alg. 1, line 5-11), and with probability p_{cross}, we either use the first parent as sibling or combine them with crossover (Alg. 1, lines 12-16). We use a custom one point crossover operator that randomly chooses with equal probability to separate the grid horizontally or vertically, as in [22]. We then apply a uniform mutation operator that can randomly mutate any gene of the chromosome with probability p_{mut} (Alg. 1, line 17). Finally, the reinsertion is elitist: during the previous steps, we generated Pop_{min} individuals, and we add the $Pop_{max} - Pop_{min}$ best parents from the last iteration P_n to the new population P_{n+1} (Alg. 1, line 24).

1 **while** *NbMax epoch not reached* **do**
2   $P_n \leftarrow$ Tournament selection of Pop_{min} $p \in P$;
3   $P_{n+1} \leftarrow \emptyset$;
4   **forall** $i \in$ [*1, size*(P_n)] **do**
5   **if** i *is even* **then**
6   j $\leftarrow i - 1$;
7   **else**
8   j $\leftarrow i + 1$;
9   **end**
10   **if** $j \in$ [*1, size*(P_n)] **then**
11   Take p_i, p_j in P_n;
12   **if** $rnd \in \mathcal{U}(0,1) < p_{cross}$ **then**
13   c $\leftarrow$ cross(p_i,p_j);
14   **else**
15   c $\leftarrow p_i$;
16   **end**
17   c $\leftarrow$ mutate(c, p_{mut});
18   l $\leftarrow$ generate(c);
19   evaluate(l);
20   c $\leftarrow$ reencode(l);
21   Add c to P_{n+1};
22   **end**
23   **end**
24   Add (Pop_{max} - Pop_{min}) bests of P to P_{n+1};
25   P $\leftarrow P_{n+1}$;
26 **end**

Algorithm 1: Genetic Algorithm

F. Level Evaluation with a Synthetic Player

To drive the genetic algorithm, we need to compute a fitness score for each generated level, depending on the gaming experience we try to provide. To do so, we use a very simple synthetic player that navigates in the level and rates it according to its preferences. For the results presented in this paper, our synthetic player performs 1125 evaluation steps to rate a 15×15 tiles level. At each step, the agent is driven by a *novelty* rating, i.e., it goes to the next adjacent cell that has been visited the longest time ago. The novelty value of each cell is computed as described in equation 2: t_n is the current time step when the synthetic player evaluates the cell, t_l is the time step of the last time the synthetic player visited this cell. t_l initial value is $-\infty$. Thus, a cell is considered a totally new for the agent if it visited it more than 200 steps ago. Our agent only walks in straight lines and can't jump, and evaluates directions in a fixed order, thus score ties always lead to the same path.

$$N = \min\left(\frac{t_n - t_l}{200}, 1\right) \tag{2}$$

To evaluate the gaming experience, we also use a *safety* score. When playing a shooter, the level geometry provides a way to hide from other players and their shots. The players can't see and react to what is happening all around them. They can thus use the geometry to their advantage to be safe on one side and shoot players from the other, for instance. Also, we think that an important aspect of the feeling provided by a level might be linked to the sight distance it provides, allowing players to shoot and to see the level from a long distance or quickly react to a player suddenly appearing in front of them in more cluttered levels. We thus compute a safety score, described in equation 3. V_{x+},V_{x-},V_{z+} and V_{z-} are the square root of the number of cells that the agent can see through in each direction, our agent moving on the xz-plane, W is the width of our level in cells, 15 in our examples. Visibility from a cell is precomputed during the level's generation, using cells' border heights and the height of the agent's line of sight, see Figure 1. We use the square root to take into account the fact that, from a perception point of view, cells close to the agent are more important than those far from it. It is to note that when running Genetic-WFC, we thus do not explicitly simulate enemy agents, but use this safety score to evaluate how geometry might provide cover against them.

$$S = \left(\frac{V_{x+} + V_{x-} + V_{z+} + V_{z-}}{W * 4} + 1\right)^{-1} \tag{3}$$

Then, we also use a *complexity* score. The idea is to evaluate how simple the level may look from a given point of view. For instance, this is very useful when generating an unsafe level. The most unsafe level, given our safety function, is a totally empty level. It is actually unsafe, but we need to express the fact that there is nothing interesting to be seen. And the same goes if the level is only filled with level one blocks, or is just a long simple corridor.

We pre-compute the complexity score as follows: when computing visibility, we use a ray that travels from the agent and goes through as many visible cells as possible. To compute complexity, we use three rays (Fig. 5). The first one is exactly the same as the visibility ray. The two others have the same length, but are one cell farther, on each side of the first one. Then, we follow these rays cell by cell, and compute complexity by testing if the module's ID of the current cell is different from the module's ID in the previous cell. If it is, it means that the players see something *different*, and thus, we add one to the perceived complexity in this direction. The sum is then divided by three, as we use three rays. We then compute the complexity score described in equation 4. C_{x+},C_{x-},C_{z+} and C_{z-} are the square root of the complexity computed along each direction from the current cell, y is our up axis, W is the width of our level in cells.

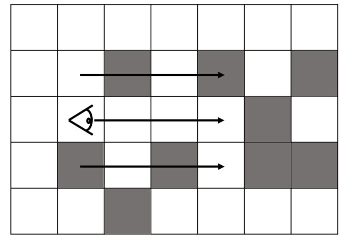


Figure 5: We compute complexity following the visibility ray, and two rays of the same length on each side of it.

$$C = \frac{C_{x+} + C_{x-} + C_{z+} + C_{z-}}{W * 4} \tag{4}$$

While the synthetic player's navigation is only based on novelty and adjacent cells' reachability, we use a weighted average of novelty, safety and complexity to evaluate the synthetic player's experience and thus calculate the fitness of each level. In our experiments, our synthetic player takes 1125 steps into the level to evaluate it, i.e. five times the number of cells in the level. The level's fitness is the average fitness for all steps.

IV. WFC Computation Cost

In order to evaluate the impact of WFC on a search based procedural level generator, we show the growth of computing time with grid size and number of modules, using our implementation of the simple tiled WFC. Our WFC is written in C# and runs in Unity game engine [23]. These tests were performed on an i7-9700k processor. Table I represents some benchmarks for the WFC computation time based on the number of modules, including air modules, and the size of the output grid. These values are an average of 100 WFC generations. We tested sizes from 15 blocks (30 meters) to 30 blocks (60 meters) and used from 2 greyblocks plus the air tile to 18 different modules. We can see that generation time grows quickly with the number of modules and grid size, and can go up to more than 300ms for a 30×30 tiles grid, with 18 different modules.

Grid Size	Number of modules			
	3	6	12	18
15×15	6	11	21	29
20×20	15	30	54	79
25×25	36	71	125	171
30×30	77	142	250	334

Table I: WFC Average Computation Time (ms)

It is to note that the number of relationships between the modules has an impact on computation time. Indeed, it can be very costly to propagate the constraints on the whole grid. If any modules can be placed next to any other, no constraints have to be propagated when we select an asset. On the other hand, if asset placement is very constrained, each choice will have to be propagated to the adjacent cells and generation should be slower. But also, the more we constrain our modules, the faster the number of possibilities will decrease and the faster the grid will be generated. The values of table I can thus vary with respect to the constraints between the modules.

Also, we can point out that to generate much bigger maps than those targeted in this paper, running a WFC on the whole map may not be efficient enough. Generation cost may be reduced by opting for a hierarchical approach. WFC allows taking into account constraints at the edge of the map, making it easy to generate a map that connects to another one. One may then generate a much bigger level as a collection of connected smaller maps. However, player experience will only be evaluated subregion by subregion. But in this example, generating with WFC four connected 15×15 maps with 18 modules should take around 120ms instead of more than 300ms for the whole 30×30 map.

In the next experiments presented in this paper, we will target 15×15 blocks levels, with 7 different modules. Level generation will be close to 10ms, which is fast enough to allow us to perform an iterative search, and provides enough variety to create different paths and spaces in the level.

V. Genetic-WFC Evaluation

We evaluate Genetic-WFC by comparing it to the approaches it should improve. Thus, we first compare Genetic-WFC with a pure genetic algorithm approach, that we will call *GA Only*. *GA Only* corresponds to a more standard, search based approach. We thus turn WFC off, and each gene of the chromosome is directly translated into an asset in the grid. We still use our synthetic player to drive the generation and thus generate levels using novelty, safety and complexity metrics. We ran both Genetic-WFC and GA Only for 6k epochs, population size were Pop_{min} = 50 and Pop_{max} = 60. Mutation probability was set to p_{mut} = 0.04, and crossover probability to p_{cross} = 0.75.

We also compare Genetic-WFC with a genetic algorithm that uses a penalized fitness function, in order to correct some asset placement errors. As we said earlier, we want our fitness function to focus on gameplay experience, but one may argue that the fitness can be calculated in two steps, one focused on experience and the other on simple level structural errors. It is however still not the same to correct these errors with optimization as it is to have an algorithm that correct these errors for us. It's either time spent in WFC or in epochs. We thus created the *GA Only Penalized* algorithm, where we remove 0.01 to the level's score for every misplaced stair and misplaced border tile, and a huge -10K penalty when there exists more or less than exactly one spawn point.

Then, we compare Genetic-WFC with a brute force search algorithm that only uses WFC to create the levels, just keeping the best level it can find, without genetic operators, that we call *WFC Only*. WFC is a procedural level generation on its own that can give very interesting results by itself, even when not driven by an optimization process. To understand how useful the genetic optimization really is and the fitness gain that it provides, we tried to generate levels using WFC only, rating levels with our synthetic player and selecting the best of them. As our GA population size is 50, and we run 6K epoch, we chose to randomly compute 6K * 50 = 300K WFC and keep only the best one.

Lastly, we wanted to illustrate how beneficial was the gene re-encoding to the optimization process, as explained in section III-E. Thus, we also ran our Genetic-WFC without the gene re-encoding.

For all these experiments, we used the same synthetic player, and its fitness function was set to $F = 1 * N + 1 * S - 1 * C$. We thus want to maximize novelt and safety, and to minimize complexity. Also, as a map is considered valid only if it contains one and only one player spawn, we put the fitness of any map with more or less than one spawn to $-\infty$.

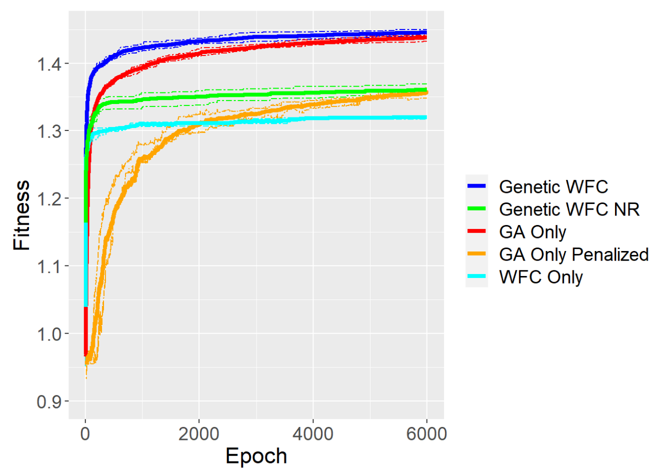


Figure 6: Average of maximum fitness evolution for 10 runs of each method, 6K epochs. Dashed lines show first and third quartiles. NR for No Re-encode. $F = N + S - C$. GA Only Penalized's fitness is reported without penalty, allowing comparison between algorithms.

	Total	Novelty	Safety	Complexity
Gen. WFC	1.45(0.006)	0.58(0.008)	0.91(0.002)	0.04(0.003)
Gen. WFC NR	1,36(0.014)	0,49(0,014)	0,92(0,003)	0,05(0,003)
GA Only	1,44(0,007)	0,56(0,006)	0,92(0,002)	0,04(0,003)
GA Only Pen.	1,36(0,014)	0,49(0,016)	0,91(0,003)	0,05(0,005)
WFC Only	1,32(0,004)	0,45(0,005)	0,92(0,002)	0,05(0,003)

Table II: Mean (sd) of scores reached after 10 runs of each method, 6K epochs, rounded to the second decimal for mean and to the third decimal for sd. NR for No Re-encode. $F = N + S - C$. GA Only Penalized's fitness is reported without penalty, allowing comparison between algorithms.

First, we can look at the fitness curve of each method, plotted in Figure 6. WFC Only's curve shows us that the genetic algorithm is actually helping WFC to reach better results. In the very first few runs, a max fitness is reached, and the fitness is almost stable for the next runs, while both Genetic-WFC and GA Only keep progressing. We thus managed to drive the WFC algorithm with a genetic algorithm to reach better results. Also, we can see that the gene re-encoding clearly helps the optimization process, as optimization is much slower when it's turned off.

Looking at Genetic-WFC and GA Only, we can see that Genetic-WFC quickly reaches a better fitness during the first epochs. This should mainly be due to the fact that WFC allows Genetic-WFC to directly sample from better levels than GA Only does. Indeed, for instance, WFC forces stairs to be placed against the walls in a navigable manner, while GA has to discover the correct placement for each set of stairs.

GA Only Penalized should be able to generate much better-looking levels, as we will see hereafter, but for the same number of epochs, the fitness reached is lower. The fitness reported for GA Only Penalized is not the full fitness used by the algorithm, but only the synthetic player perception part of it, in order to be able to compare the values.

It is also to note that we compare algorithms for a fixed number of 6K epochs. Genetic-WFC spends around 9.5 minutes for 6k epoch in our setting, while GA Only and GA Penalized need only around 2.5 minutes, which is almost 4 times faster. We thus ran GA Only Penalized for 10 runs of 22K epochs to reach the same computation time. The best score for ten runs was 1.42, which is better than 1.38 but still lower than the 1.457 we achieved with 6k epochs of Genetic-WFC.

Table II allows us to have a better understanding of the performance of each method: we give the value of each parameter of the fitness function for the best maps created in 6K epochs and presented hereafter. We can see that the GA Only slightly outperformed Genetic-WFC for Safety and Complexity, we suppose mainly by staying low and using many stairs. However, this strategy has a limit as the player needs to go back after going on any unconnected stairs tile, therefore penalizing the novelty score.

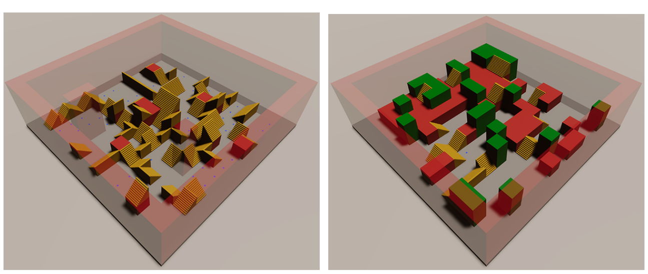


Figure 7: Best levels for 10 runs of 6k epochs, maximizing novelty and safety with the lowest complexity. On the left: GA Only. On the right: GA Only Penalized.

If we look at the best-rated maps, we can better understand the relative performance of each method. Fig. 7 shows the GA Only's best results. We can see that GA Only placed a lot of stairs. Stairs are very useful to both optimize for novelty and complexity: stairs are navigable and can thus provide positive feedback for novelty, but they also block visibility and can help to maximize safety. We can also note that GA Only used border modules inside the map, as nothing prevented it to do so. However, even if this map has a correct fitness, the map is clearly not the kind of level we want to generate. It is not the case for GA Only Penalized, which has a much better look. We still have some asset placement errors, as we only penalize them and do not correct them.

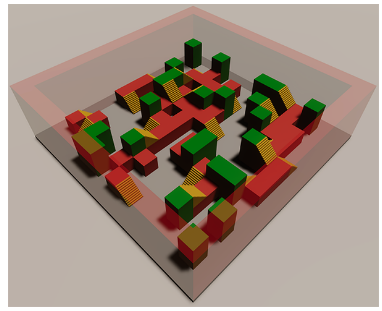


Figure 8: WFC Only's best of 10 runs of 6k epochs, maximizing novelty and safety with the lowest complexity.

As we can see in Figure 8, WFC Only gave a better-looking result than GA Only. However, the map's fitness is much lower than for the GA Only and Genetic-WFC best results. If we look more precisely at the scores given in table II, we can see that WFC only was really penalized by novelty. Indeed, our rules allow creating locally navigable portions of levels, as stairs are connected correctly for instance, but making a level that is globally navigable is much harder. Of course, we might change the modules' placement rules to disallow any connection that breaks navigability, by for instance forcing any level one tile to be connected to another level one tile or to a set of stairs for instance. But then, it would then be much harder for the generator to constrain the player's path in order to provide a specific experience.

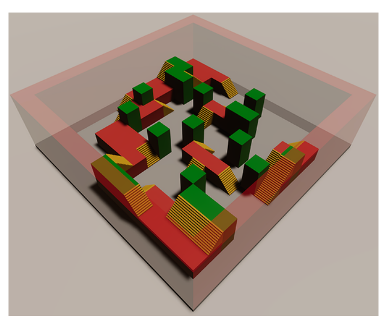


Figure 9: Genetic-WFC's best of 10 runs of 6k epochs, maximizing novelty and safety with the lowest complexity.

Finally, if we look at Figure 9, we can see that Genetic-WFC did, in our opinion, create the best-looking map with regard to the constraints. The novelty score is the highest, and there do not exist any level one or floor tile that can't be attained. Few level 2 tiles are reachable, but they are close to the border wall, limiting the impact on safety and complexity. Other level 2 blocks are scattered around the level, blocking sight and thus providing safety and limiting complexity.

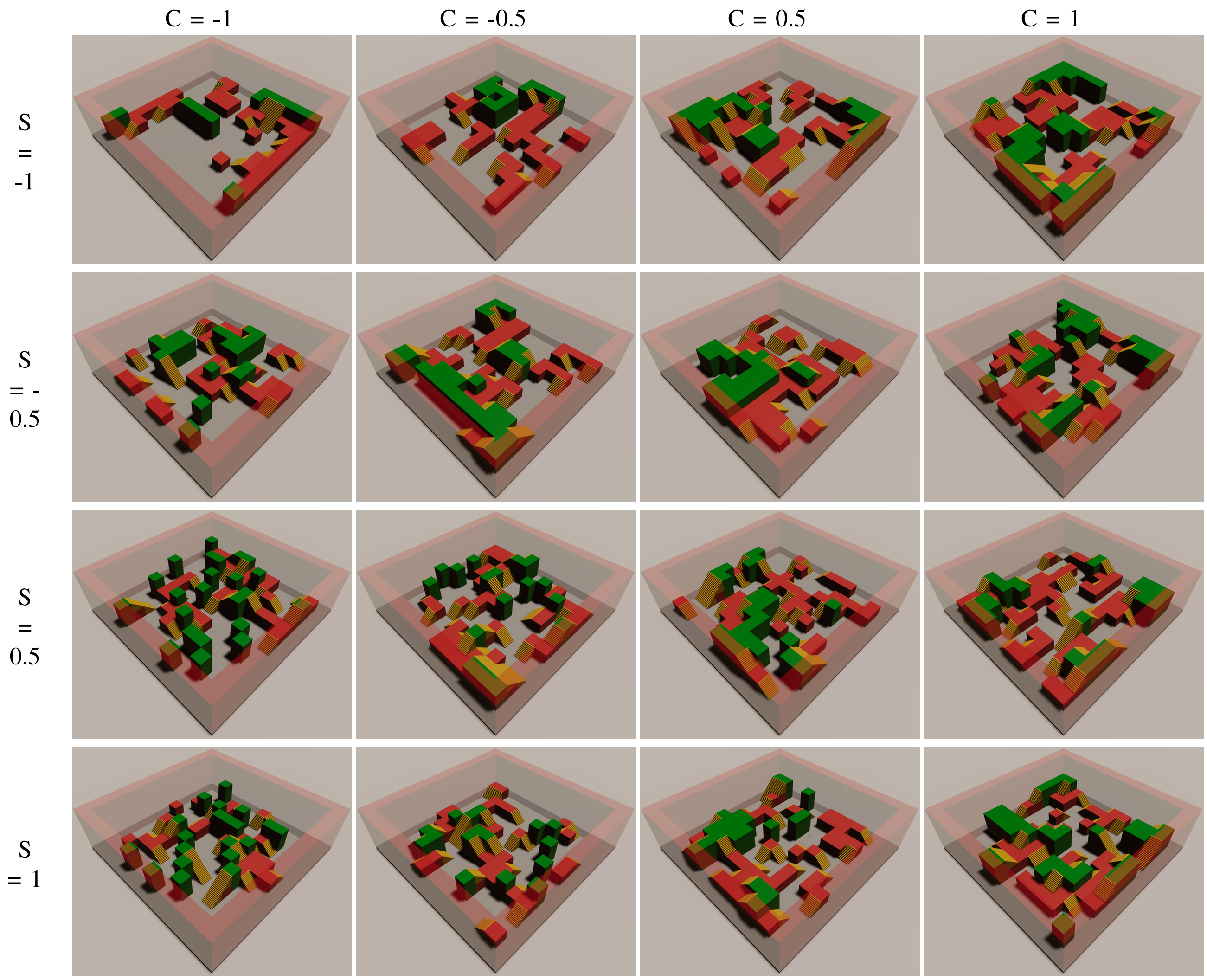


Figure 10: Generated Levels with varied safety and complexity, and novelty weighted 0.5

VI. Exploring Game Levels' Space

In order to evaluate the variety of experiences that can be provided by our generator, we decided to sample the level space and use Genetic-WFC to create maps with different values of novelty, safety and complexity. We cannot depict here all the results that can be obtained, so we limit ourselves to the following. Novelty is a parameter that we chose to keep at a fixed weight of 0.5 for all the runs. Indeed, optimizing for low navigability can be done, but as we cannot explore all dimensions here, we chose exploring for varied levels of complexity and safety while maintaining navigability was an interesting setup. We used four possible weights for complexity and safety: high (1.0), medium high (0.5), medium low (-0.5) and low (-1.0). We ran Genetic-WFC for all parameters combinations for 5k epochs, with population size Pop_{min} = 50 and Pop_{max} = 60. Mutation probability was set to p_{mut} = 0.04, and crossover probability to p_{cross} = 0.75. We show the results in Figure 10.

Looking at Figure 10, we can observe that both complexity and safety do have an impact on the generated maps' layout. For instance, in the first row, we try to generate maps with a very low safety scores. Such maps can be very empty, like the top left one, as the player can easily navigate an empty map, and is unable to hide if there is nothing but air. However, as we go to the right, from low to high complexity, the generator adds more and more modules, creating both unsafe and cluttered maps. To do so, we can see for instance that all the level 2 modules are navigable, providing positions with low safety and high complexity.

Then, if we look at the first column in Figure 10, we see that starting from the top left, a relatively empty unsafe and non-complex map, then if we look farther down Figure 10, safety is increased while keeping the complexity low. Many of the level 1 tiles are navigable, but the generator adds level 2 tiles that are not, providing cover while maintaining safety.

The right column of Figure 10 is harder to interpret. As we go down, we ask for something rather contradictory: we want safety to increase but also want to maintain a high level of complexity. A map that maximizes complexity and safety both allows the player to see far away to see complex structures and limits visibility to keep a high level of safety. The only difference between the top and bottom maps we might spot is

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2022.3192930

that the bottom level one blocks seem to be placed in a more narrow way than in the top map. Indeed, in the bottom, for the safest map, level one blocks only provide paths of one block width, i.e., we can't see level one blocks that create a platform of two tiles by two tiles or more. On the other hand, at the top, for the most unsafe map, this is not the case, and we see larger level one platforms. Avoiding platforms might be a way for the generator to gather fitness with safety on these narrow paths, as most of them are also close to geometry. Thus, the bottom right map is very interesting as it seems very varied, with unsafe and complex positions at the top of level two tiles but also safe paths on level one tiles and floor tiles.

VII. Conclusion and Future Works

We presented Genetic-WFC, a procedural level generation algorithm that combines a genetic algorithm with the Wave Function Collapse algorithm as a repair operator, to generate levels targeting specific play experiences. WFC allows us to only manipulate levels that respect basic placement constraints and to focus our fitness function and simulation steps on gaming experience. We control WFC by locally biasing asset selection probabilities. We use a specific 2D crossover operator to split the map vertically or horizontally. We also re-encode genes by using the actual modules chosen by WFC in the generated level, in order to maximize crossover efficiency. We propose a Greyblocking step to limit the combinatorial explosion of WFC.

We show that we outperform a pure genetic search and a brute force search with WFC only, in terms of level fitness. Having better results than a brute force search shows that the genetic algorithm actually controls WFC. For this gameplay and this setting, Genetic-WFC seems to be the best algorithm. With the same number of steps it reaches the highest fitness level, and with the same amount of time our algorithm is just slightly better than Penalized WFC. However, our implementation of WFC currently runs in C# and might still be optimized in order to run faster, which is not the case for the penalized approach.

We used a synthetic player with very simple metrics of novelty, safety and complexity. We showed that from a bird's-eye point of view, maps generated by sampling various weights of safety and complexity seemed to have different features that match the game experience targeted by each set of metrics weights. This confirms that a simple simulation of the map can give interesting results, that our metrics seem to propose useful dimensions, and that Genetic-WFC may be able to provide various play experiences.

We may be tempted to calculate the levels' fitness without simulation, by averaging the safety and complexity scores for all cells. However, the gameplay we target is emergent, and the player will not spend the same amount of time in every part of the level. We need to take into account available paths, evaluate where the player will be the most, the direction they are facing, which is getting close to actually simulating the player's journey. Moreover, we do not currently take actual enemies into account, but the simulation approach makes it much more straightforward to do than to statically evaluate the possible impact of enemies patrols on the resulting gameplay using heuristics.

The next step of this research is to perform a user evaluation of our generated levels. To do so, we need to explicitly simulate the enemies, and thus provide human players with a more specific context. Then, our synthetic player might also be improved by using a cone of perception instead of a straight line, and allowing it to fight with enemies, for instance. Also, we may further investigate the advantage of our method by comparing it to constraint based optimization methods like FI-2Pop, and see how they perform on generating levels for our test-bed gameplay.

References

[1] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016, website: http://pcgbook.com/. Last visited 30 Jun. 2021.
[2] M. Kleineberg, "Infinite city," https://marian42.de/article/wfc/, 2019, accédé le 25/10/2019.
[3] PlausibleConcept, "Bad north," 2018.
[4] O. Stålberg, "Townscaper," 2020.
[5] Ubisoft, "Ghost recon: Wildlands," 2017.
[6] ——, "Far cry 4," 2014.
[7] F. Software, "Elden ring," Bandai Namco Entertainment, 2022.
[8] M. Gumin, "Wave function collapse," URL: https://github.com/mxgmn/WaveFunctionCollapse, 2016, last visited 3 Oct. 2019.
[9] R. Devaux, "Week 60: Lots of things," URL: https://trasevol.dog/2017/06/19/week60/, 2017, last visited 16 Jan. 2020.
[10] A. Wallace, "Maureen's chaotic dungeon," URL: https://globalgamejam.org/2019/games/maureens-chaotic-dungeon, 2019, last visited 25 Oct. 2019.
[11] FreeholdGames, "Caves of qud," 2015.
[12] T. Nordvig Møller, J. Billeskov, and G. Palamas, "Expanding wave function collapse with growing grids for procedural map generation," in *International Conference on the Foundations of Digital Games*, 2020, pp. 1–4.
[13] A. Sandhu, Z. Chen, and J. McCoy, "Enhancing wave function collapse with design-level constraints," in *Proceedings of the 14th International Conference on the Foundations of Digital Games*, 2019, pp. 1–9.
[14] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.
[15] S. Dahlskog and J. Togelius, "Procedural content generation using patterns as objectives," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2014, pp. 325–336.
[16] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, and G. N. Yannakakis, "Multiobjective exploration of the starcraft map space," in *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*. IEEE, 2010, pp. 265–272.
[17] A. Liapis, C. Holmgård, G. N. Yannakakis, and J. Togelius, "Procedural personas as critics for dungeon generation," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2015, pp. 331–343.
[18] C. Holmgard, M. C. Green, A. Liapis, and J. Togelius, "Automated playtesting with procedural personas with evolved heuristics," *IEEE Transactions on Games*, 2018.
[19] A. Liapis, G. N. Yannakakis, and J. Togelius, "Constrained novelty search: A study on game content generation," *Evolutionary computation*, vol. 23, no. 1, pp. 101–129, 2015.
[20] S. O. Kimbrough, G. J. Koehler, M. Lu, and D. H. Wood, "On a feasible–infeasible two-population (fi-2pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch," *European Journal of Operational Research*, vol. 190, no. 2, pp. 310–327, 2008.
[21] D. Giacomelli, "Geneticsharp," URL: https://github.com/giacomelli/GeneticSharp, 2013, last visited 2 Jul. 2021.
[22] M.-W. Tsai, T.-P. Hong, and W.-T. Lin, "A two-dimensional genetic algorithm and its application to aircraft scheduling problem," *mathematical Problems in Engineering*, vol. 2015, 2015.
[23] Unity, "Fps microgame," URL: https://learn.unity.com/project/fps-template, last visited 25 Mar. 2021.