Castle on the Hill: Evolving Castles in Difficult Terrain using Quality Diversity

Jakob Ehlers & Oliver Hansen
Artificial Intelligence for Games and Simulations
KGARIGS1KU
IT University of Copenhagen

1 Introduction and problem statement

From Dracula's Castle (Konami 1986) to Princess Peach's Castle — castles and fortifications have been a staple in video games, both as a backdrop and as interactable levels.

The form and layout of the castle have primarily been aimed at optimizing either of these purposes, rather than optimizing for the function of an actual castle: Defending an area and repelling attackers.

How well a castle functions as an actual stronghold is likely less important in games such as *Super Mario 64* (Nintendo EAD 1996), since the idea of an actual siege by Bowser and his lackeys might be a stretch of the imagination. Prioritizing aesthetics over functionality can certainly be the sensible approach.

However, for a more complex game that is grounded more in reality, such as *Mount & Blade II: Bannerlord* (TaleWorlds Entertainment 2022), the player might start to wonder how exactly this fortification is supposed to function as a defensive structure.

Creating these castles for the purposes of video games have usually been the job of level designers and artists. Typically, the goal for a level designer has an emphasis on exploration and playability, while for an artist the emphasis is on making it aesthetically pleasing and fitting to the setting. Neither of these roles tend to have the goal of creating feasible defensive structures. This is where machines may be of help.

Procedural Content Generation (PCG) is "the algorithmic creation of game content with limited or indirect user input" (Shaker, Togelius, and Nelson 2016). Content is most of what is inside a video game, including levels, dungeons, and even stories and game systems.

Generating complex architectural structures for games can represent an interesting challenge for PCG. It must be able to create a navigable structure that is interesting to explore and fits the setting of the game.

Castles were usually not constructed independently from terrain features, such as hills, mountains, lakes and rivers. Often, these features provided defensive advantages, as they could be used as natural obstacles and make the approach more difficult for attackers (Dusterwald 2015).

In some video games, however, terrain primarily functions in a visual or navigational context, rather than as a factor that directly shapes the architectural design of the castle itself.

Castles that meaningfully conform to these terrain features go beyond the standard design challenges. There are many factors such as attacking routes, choke points, natural barriers, and elevation changes. For a designer to take all of this into consideration risks that they overly rely on their assumptions of what a good fortification is.

A well-defended castle next to a lake may look vastly different than one next to a river or in a mountain range. Therefore, a generative algorithm should search for many different high-performing solutions.

Previous work on procedural content generation of castles and similar structures has had a focus on historical accuracy and believability (Mas, Martin, and Patow 2020). Other studies have used grammars and L-systems (Parish and Müller 2001). This paper aims to expand the field by using search-based procedural content generation to build up a diverse archive of high quality solutions that try to conform to a set of terrain features.

This paper implements a quality diversity algorithm called Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) (Mouret and Clune 2015) in order to explore how battle simulations and evolutionary algorithms using quality diversity can procedurally generate castles that conform to terrain features, such as mountains and lakes. It will explore which behavioral dimensions yield the best archive of varied, but well-defended castles. As a baseline, the results from the quality diversity algorithm will be compared with results from a conventional evolutionary algorithm, which only seeks to optimize for fitness.

In the remainder of this paper, we will review related work in the field, introduce our battle simulation and MAP-Elites implementation and present and discuss the results of our experiments.

The project was version controlled on GitHub, and the repository can be found on this link: `https://github.com/OliFryser/AIGS-Castle-Generation/`

2 Related Work

2.1 Procedural Generation of Cities and Castles

Parish and Müller explored procedural generation of geometry with grammars using *L-systems*, creating entire roadway layouts for cities from a set of rules (Parish and Müller 2001), as well as buidling facades within these cities. The L-systems are *self-sensitive*, in that they respond to some constraints that can extend streets or slightly move them, so that they intersect in the intended manner.

Müller et al. developed a *shape grammar*, called CGA Shape. They point out that while L-systems are excellent for generation of use-cases such as road-way layouts and trees, they overemphasize growth so that it becomes counterproductive in the procedural modelling of buildings (Müller et al. 2006). They present a notation for production rules that can be both deterministic and stochastic.

For our study, L-systems would likely have resulted in an overemphasis of growth of the castle. Therefore, we decided to develop our own grammar to represent the layout of our castle. This enables an interface to stochastically vary the castles, as well as save and load them to and from a text file. This grammar will be discussed in later sections.

Mas et al. simulate the evolution of ancient fortified cities (some of them being castles) with a focus on historical accuracy (Mas, Martin, and Patow 2020). They use battle simulations to evolve the defence. Once a wall is breached by the attacker, it is improved upon according to a set of rules. They simulate city growth, castle expansion and changing requirements over time, due to advancements in military technology. Our approach similarly makes use of battle simulations for performance evaluation of castles, yet applies them in a more abstract and simplified environment.

The study by Mas et al. uses the term *evolution* to describe how ancient cities evolve over time. In contrast, our approach uses evolution in the context of evolving solutions from previous solutions in the algorithmic sense.

In Dusterwald's thesis, the focus is on placing castles in a procedural terrain (Dusterwald 2015). The terrain is a 3D voxel-based terrain, similar to *Minecraft* (Mojang Studios 2011). A mixture of different noise functions are used generate the terrain. Afterwards, a voxel castle is placed, such that it looks natural in the procedural terrain.

Their project is ideal for a use-case like Minecraft, where a vast procedurally generated world may need to have many castles and others structures placed in a sensible manner.

In our study, we start by placing a target and then grow the castle around it. In this sense, the use-case analogy is having a place one would like to protect and then growing fortifications around that, similar to how castles would form around cities (Mas, Martin, and Patow 2020).

Dusterwald uses a brute-force search for the optimal fitness within the specified area, meaning that a castle is deterministic in the way it is generated on a terrain, and uses height maps to calculate the fitness. Our study uses an evolutionary search approach, with a lot of random variation, meaning results between runs differ significantly. Further, a brute-force search would not have been feasible, as the castle has too many variations to be exhaustively searched through.

Dusterwald uses random terrain, and for the maps in their results, they hand-picked about 30 maps with interesting features. Dusterwald notes ”Stronger variation in the terrain also meant stronger variation in the castles produced” (Dusterwald 2015).

We did initially implement Perlin noise terrain, but decided against using it. Instead, we hand-crafted maps with more extreme features using a tool we created called *TerrainBuilder*. We emphasized extreme terrains for the same reasons as Dusterwald points out.

2.2 Castles and Terrain

In the introduction, we mentioned examples of games, where castles do not adapt to the terrain, or use the terrain features defensively. We will now discuss a counterexample.

The recent game *Kingdom Come: Deliverance II* (KCD2) (Warhorse Studios 2025) takes place in a region of current day Czech Republic called ”Bohemian Paradise”. This region features a castle that *does* adapt to the terrain, taking advantage of the high ground afforded by two hill tops with a lower castle connecting the towers.

Of course, KCD2 is inspired by the real historical location. Figure 1 shows an image of the actual Trosky Castle. Note how the castle construction is adapted to the terrain.

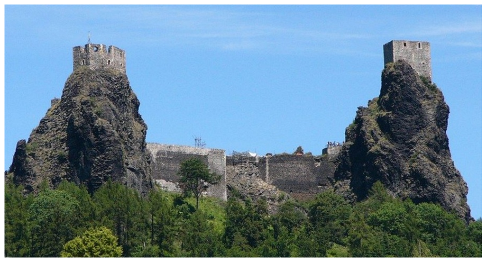

Figure 1: Image of Trosky castle (Interregion.cz 2018).

Dusterwald also emphasizes how the terrain has affected the shape of some welsh castles (Dusterwald 2015).

We mention this to highlight, how taking inspiration from real world castles can lead to castles that adapt to the terrain, since this is how castles are often built in the real world. In the following sections we will discuss how one might create such castles algorithmically.

2.3 Search-based Procedural Content Generation

Procedural Content Generation (PCG) has many purposes. One of them is to spark the imagination of designers and artists, making them consider of new ways of producing a certain piece of content (Togelius et al. 2011). The purpose of our study is not to produce castles that are ready for production. Rather, the point is to provide inspiration to designers and artists concerning how one might construct a castle in a certain terrain.

In their survey, Togelius et al. define a taxonomy of PCG (Togelius et al. 2011). One distinction is drawn between *constructive* and *generate-and-test* approaches to PCG, where the former constructs something once according to some rules, and the latter generates something, tests it and discards it and tries again if it fails.

Search-based PCG is a specific kind of generate-and-test, where instead of simply discarding or accepting a candidate, it is evaluated with some kind of *evaluation function*, which produces some value, often called *fitness*. A lot of search-based PCG is based on some form of *evolutionary* algorithm, where the population is evolved from the best of the previous iterations (Togelius et al. 2011).

Usually the content is generated in some format that can be *expanded* by some algorithm into the actual piece of content in the game. This is referred to as an *encoding*. In search-based PCG, the generated format is called the *genotype* and the expanded content is the *phenotype*. This encoding (or mapping) between genotype and phenotype can be *direct* or *indirect*. With an direct encoding, the mapping from genotype to phenotype is relatively simple. With an indirect encoding, the mapping is often very complex (Togelius et al. 2011).

In our project, the genotype is a grammar, and the phenotype is a castle on a grid. This is an indirect encoding.

The *locality* of a genotype describes how changes in the genotype affect the phenotype. For representations with *high locality*, this means that small changes to the genotype will result in small changes to the phenotype and small changes to the fitness. According to Togelius et al., for a genotype to be a good representation it should have high locality. Further, all interesting solutions should be representable by the genotype, meaning that no solutions are ”overlooked”, because the genotype cannot represent them (Togelius et al. 2011).

How well our genotype works for these criteria will be discussed in the section 6.

On the topic of evaluation functions, Togelius et al. defines three classes: Direct, simulation-based, and interactive (Togelius et al. 2011). Our evaluation function is simulation-based, as it is based upon the results from the battle simulation.

2.4 Quality Diversity and MAP-Elites

Many PCG algorithms promote either the quality of the solutions *or* the diversity of solutions. A family of evolution-like PCG algorithms, called *Quality Diversity* (QD) algorithms, promotes both at the same time. They reward divergence from the other candidates, while promoting quality (Gravina et al. 2019).

One of the motivations for QD algorithms is that they can produce many high-performing individuals that are very different from each other. They also provide good opportunities for *mixed-initiative* content design, since designer get to see many different high-quality solutions they may not have thought of (Gravina et al. 2019).

These qualities align with our purpose of sparking the imagination of designers and artists.

To promote diversity, QD algorithms need to describe the *features* (or *behaviors*) of an individual. This defines what behaviors diversity is wanted across. The behaviors can be interpreted as an N-dimensional space, where each dimension is one of the desired behaviors, referred to as the *behavior space* (Gravina et al. 2019).

To promote quality, QD algorithms can use *local competition*, letting individuals that are close in behavior space compete, or *constraints*, determining whether an individual is *feasible* or *infeasible* (Gravina et al. 2019).

One quality diversity algorithm is *Multi-dimensional Archive of Phenotypic Elites* (MAP-Elites), where the behavior space is partitioned into a multidimensional grid, called the *archive*. Each cell holds the best performing individual that fits into that part of the behavior space (Gravina et al. 2019).

In the introductory paper, Mouret and Clune describe MAP-Elites as an *illumination algorithm*, since it illuminates a search space that the user gets to define (i.e. the behavior space) (Mouret and Clune 2015). They find that in addition to illuminating the search space, it also finds a better performing solution than more traditional evolutionary algorithms, and even other QD algorithms such as *Novelty Search + Local Competition*.

Our goal is to find castles that are well-defended, yet also conform to the terrain and uses those features defensively. By choosing to use MAP-Elites as our algorithm, we expect to find that at least some of the high-performing solutions in the archive will have these characteristics.

3 Environment Description

The environment we use for generating castles and testing them for fitness is a python-based proprietary system dubbed *Fortify*. Fortify is a simple simulation system where one attacking team attempts to storm a castle while a defending team attempts to repel them. The basic components are: A simulator, a level, and units.

A level is created from a terrain map consisting of a height map and lists of coordinates for static terrain features such as water and a path that leads to a point that simultaneously indicates the target for the attackers and the area the castle should defend. Another map representing a castle floor plan can then be placed over the terrain. From these, a *graph* is computed.

Units navigate the graph using the $A*$ path-finding algorithm, and their behavior is controlled by a *hierarchical finite state-machine*. Units are slower on steep terrain, and are not allowed on water. Units have a type, and the unit types are *Axeman* and *Archer*. Axemen are on the attacking team, and will move towards the target and break down gates that block their approach. Archers are on the defending team, and will shoot at enemies within their range. When a unit reaches 0 hit-points, it will be removed from the simulation.

Before a run, the simulation will prepare by first cleaning up any castle elements and units from last run. Then it will tell the level to create a new castle by providing the data to build a castle in the form of a tree of castle instructions. Each *node* in the tree contains a list of *instruction tokens*. The instruction tokens implemented for this project are as follows: **WALL**, **TOWER**, **EMPTY**, **LEFT**, **RIGHT**, and **BRANCH**.

A castle generator will place a special castle segment, the *keep*, just above the target provided by the level. This is the starting point of the castle. The generator will then pass the instructions to a castle generation *agent*. The agent will start on the keep facing north. One-by-one it processes the instructions. If encountering either a **WALL** or a **TOWER**, the agent takes a step forward and places a castle element. If encountering an **EMPTY**, it steps forward and clears the new position. Alternatively, on receiving **LEFT** or **RIGHT**, it will change its facing by turning 90 degrees counter-clockwise or clockwise respectively. Finally, if it encounters **BRANCH**, it will spawn another agent that inherits its direction and is given a sub-tree of instructions that it will start parsing the same way.

When the agents have placed all the segments, gates will placed where castle segments intersect with with the path, in order to ensure access to the target courtyard.

After the castle has been created the simulation organizes units into two teams, attacker and defender, the defending team consists of an archer placed in the middle of each castle tower section, and the attacking team consist of eight axemen placed around the edges of the map in accordance with the eight winds. For each archer given to the defending team, an additional axeman is given to the attacker and placed alongside already placed attackers in a clockwise manner. The simulation will run until an attacker reaches the target, all attackers have been killed, or until the simulation has run for 40000 steps.

4 Methods

4.1 MAP-Elites

We have implemented the MAP-Elites algorithm as our search and optimization algorithm. The high-level idea is to maintain an archive of high-quality solution, where this archive is a multi-dimensional grid, and each cell corresponds to a partition of behavior space. Gravina et al. points out how a fundamental challenge of QD is working with high dimensionality of behavior spaces and figuring out how to partition the behavior space (Gravina et al. 2019).

For this reason, we used a 2-dimensional behavior space, where each dimension has 10 cells. This enables us to visualize the archive as a 10×10 grid of castles. Further, as we examine the castles manually, a total of 100 castles per run keeps the dataset to a manageable size. Exactly which behaviors we have used will be discussed later in this section.

We used the castle instruction tree as the genome. It is compact and it allows for relatively straight-forward crossover and random sampling. Using the castle generator it can be mapped to its phenotype, the actual castle in the map.

Each instruction tree has a root, which is a *node*. Each node has other nodes — its children. Apart from this, the instruction tree maintains a list of all of its node in every sub-tree to allow for random sampling in constant time.

When MAP-Elite starts, it initializes a starting population by randomly generating solutions. Each *individual* in this population is evaluated with the *evaluation function*. The evaluation of an individual yields its *behavior* and *fitness*. We then discretize the behavior to get a *key* to the archive in some way. This maps the observed behavior from a continuous behavior space to our partitioned behavior space. If the entry for this key in the archive is empty, the individual is saved to the archive. Otherwise, if there already is an entry, the fitness of the old entry and the new individual are compared. If the new individual has a higher fitness, it is saved in the archive. Otherwise, the old entry is kept. This is the *local competition* that optimizes for quality in the archive.

The user can specify the amount of individuals in the initial population by changing the **population** parameter in the configuration file. The random generation of individuals add between 10 and 50 random instructions to the instruction tree.

The random sampling of instructions is weighted according to a set of *mutation weights*. There are different mutation weights for initial sampling and for random variation of existing instruction trees. These mutation weights control, how likely it is that a certain instruction is picked. A lot of time could be spent on the task of identifying good weights, but ultimately, it likely matters more how many iterations of the algorithm is run, rather than the tuning of the mutation weights. Therefore, we did not spent a lot of time tweaking these weights. Better results may be achievable by finding the optimal values for these.

Nevertheless, the initial sampling favors walls more than the random variation sampling, while disfavoring towers and empty spaces. In this way, we get a lot of walls, and hence structure, initially. The disfavoring of towers also result in less attackers, since the attackers are proportional to the amount of towers. This gives the initial population a greater chance of surviving for longer, since there is more structure and less attackers.

After the population is initialized, a user-specified number of **iterations** are run, where a random individual in the archive is sampled, randomly varied and then *evaluated* with the evaluation function. This yields us the new behavior and fitness of the individual.

The tree supports four types of variations: *Additive mutation*, *destructive mutation*, *substitutive mutation*, and *crossover*. The type of variation is chosen randomly, where crossovers and destructive mutations are slightly more likely than the others. This is based on initial experiments, and seemed to provide a good balance of growth and keeping the castles to a manageable size.

The additive mutation works by sampling a random *node* in the instruction tree. Then, it selects a random instruction tree token, using the mutation weights for varying, rather than the ones for visualizations. If a branch instruction is inserted, a new *node* is added to the sampled node's children. To ensure the branch is inserted in the correct position in the list of children, the amount of branch instructions before the inserted instruction is counted.

The destructive mutation samples a random node in the instruction tree. Then it selects a random *mutation index*: The instruction to be destroyed. If it is a branch instruction, the corresponding child is also removed in a similar way to additive mutation of branch instructions.

The substitutive mutation is essentially a combination of the destructive mutation and additive mutation, as we destroy an instruction, then inserts a new one at the exact index that we destroyed it.

Crossover is done by sampling another individual from the archive randomly. Then a node is sampled from both of the individuals, dubbed *new parent* and *sub tree*. The sub tree is then inserted under the new parent, replacing one of the original children of the parent, if it has any.

After variation, the resulting individual is mapped to its phenotype, through the *castle generator*. Then the phenotype is evaluated through a run of the simulation. This yields the fitness and behaviors of the castle, which is then used to decide whether or not it should be saved in the archive, according to the same logic as for saving the initial population to the archive.

We experimented with different behaviors for the behavior space. We also experimented with different fitness functions. We did this through a number of tracked variables. Each tracked variable is stored in the *state* of the simulation. These can be combined pairwise to define the 2-dimensional behavior space. The following variables are stored:

- *Blocks*: The amount of blocks used.
- *Cost*: The weighted cost for blocks, e.g. towers are more expensive than walls.
- *Area*: The total area enclosed by the castle.
- *TowerRatio*: The ratio of the amount towers over the amount of walls.
- *Kills*: The amount of attackers shot by towers.
- *Gates*: The amount of gates generated by the path through the castle.
- *Towers*: The amount of towers.
- *East-West Ratio*: The number of blocks to the east of the keep, divided by the sum of blocks to the east and the west of the castle.
- *North-West Ratio*: The number of blocks to the north of the keep, divided by the sum of blocks to the north and the south of the castle.

For partitioning the behavior space, we used a *dynamic key*. The key is calculated through a class called **DynamicCeiling**. In this class we have three member variables. A *maximum*, setting a hard maximum for the behavior, meaning anything above will be thrown in the last behavior space partition. Then a *floor*, which can set a minimum, so that anything below will be thrown in the first space partition. In practice, *floor* was never used. Lastly, a *ceiling*, which starts at a default of 1 and increases when an individual is seen with a value greater than the current *ceiling*, but less than the *maximum*. The key is then calculated as:

$$key(value) = round((value - floor)/ceiling) \cdot indices$$

The indices was set to 9, as we wanted 10 cells for each behavior, ranging from 0 to 9.

Usually one would partition the behavior space with some set maximum, and use that instead of our ceiling. This makes sense for some of our behaviors, such as the east-west ratio, since this behavior ranges from 0 to 1.

However, for some behaviors, such as blocks, a hard maximum is more difficult to define. Of course, the actual maximum would be simple enough to compute. It is the maximum number of blocks that can fit in the map, which for a 120×120 size grid, where each castle block is of size 5×5 and there is a padding of 10 units on every side, leaves us with $((120 - 20)/5)^2 = 400$ total blocks. Setting this as the maximum, however, would result in mostly the lower part of this behavioral dimension being explored, at least until the castle grows large enough. We want to relatively quickly fill the archive as much as possible to enable more crossover and optimization of new solutions, so this is not desirable. Therefore, the dynamic ceiling approach enables the archive to fill up, and then readjust.

This has some performance implications though, as the archive will have to *shift*, every time the ceiling increases. One way to view this, is that the partitions start off smaller and fine-grained and then slowly become larger and more coarse. Therefore, two individuals that were previously in two separate partitions may fall into the same partition after increasing the size of the partitions. As the archive has at most 10 individuals along one dimension, the performance implications are negligible.

It is worth noting that some areas of a behavior space cannot be explored at all, if the two of the dimensions are dependent upon each other (Gravina et al. 2019). In our case, two dimensions that were dependent were *area* and *blocks*, as it is impossible to create a large area without many blocks, but also not possible to have a large area with too many blocks, since the blocks themselves would fill the area.

Other approaches to behavior space partitioning have also been explored, such as *MAP-Elites with sliding boundaries* (Fontaine et al. 2019). However, this was not pursued for this project.

We tried multiple fitness functions, but settled on two. One fitness function is simply the *step count* of the simulation, i.e. the amount of steps from start to end of the simulation. Castles that survive the battle for longer has a higher fitness.

The other fitness function addresses a problem we experienced with castle growth. As Müller et al. remarked, L-systems have an overemphasis on growth (Müller et al. 2006). It seems as though our grammar and the way it is grown show similar characteristics. To try to punish the castle for just growing without any purpose, a second fitness function also uses the step count as a baseline, but then also takes the enclosed area, cost and kills into account.

To have something to compare our findings of MAP-Elites with, we implemented another algorithm, which does not make use of behavior spaces.

4.2 Conventional Evolutionary Algorithm

The conventional evolutionary algorithm (CEA), we used as a baseline, is a standard genetic algorithm (Mitchell 1998). It was only slightly modified to match the existing genome and system architecture:

1. Initialize population.

2. Evaluate population and select prospects for random variation.
3. Randomly vary the prospects until enough to fill out a new population has been created.
4. Replace old population with new population.
5. Go to step 2.

This would run in batches of 500 generations, with a population of 20, on the same maps as the MAP-Elites. In order to stabilize the progress, an elite strategy was employed that conserved 10% of the best prospects from each generation.

4.3 Running the experiments

When running the MAP-Elites, the *coverage* (amount of cells filled in the archive), *max fitness* (the fitness of the best performing individual), and *QD-score* (the average fitness within the archive) is saved in each iteration. These can then be visualized on plots. Further, the archive is visualized with the entry's fitness

When running the CEA algorithm, only the max fitness is recorded. The resulting individuals are visualized in the same archive format. However, the cells are not mapped to behaviors.

The experiments were run on two maps with different terrain features. In the following section, we will present the results.

5 Results

For both maps (see figure 2), MAP-Elites was run for 2500 and 5000 iterations. We tested two behavior spaces. One with *East-West Ratio* as the x-axis and *North-South Ratio* as y-axis, and another with *Cost* as the x-axis and *Area* as the y-axis. The first of these behavior spaces will be referred to as SNEW, and the second as AC, for the sake of brevity. SNEW uses the more complex fitness function to limit the growth of the castle. AC uses the simple fitness function, since cost and area is taken into account through the behavior space.

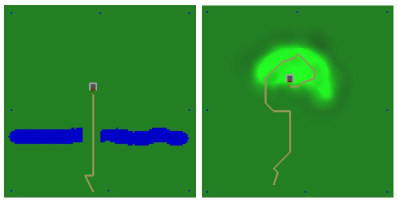

Figure 2: Map 1 (left) and Map 2 (right) that was used for the experiments

CEA will be run for 500 iterations with a population of 20 for both maps. It uses the complex fitness function.

5.1 Map 1

The first map is a flat map with a river intersecting the bottom part, and a ground path crossing the river, the river does not go all the way to the edges in order to not completely cut off access to the target.

Comparing the MAP-Elites results for SNEW and AC, a good metric for how well the behavior space is defined is the coverage. Looking at figure 3, it can be seen that the SNEW reaches max coverage (100) faster than AC. Note that the coverage for AC decreases suddenly when the archive shifts. Even with 5000 iterations, AC ends with about 50 % coverage. This implies that SNEW exhaust the search space quickly and can spend more time optimizing for fitness.

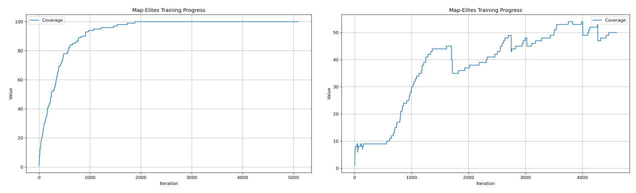

Figure 3: SNEW (left) and AC (right) coverage for a run with 5000 iterations for map 1

Comparing the QD scores of running SNEW for 2500 and 5000 iterations respectively, (see figure 4) they both incline steadily throughout, this makes sense as the behavior space fills out and better solutions are found, however the flattening of the Max fitness hints that the QD-score will also flatten as the scores converge.

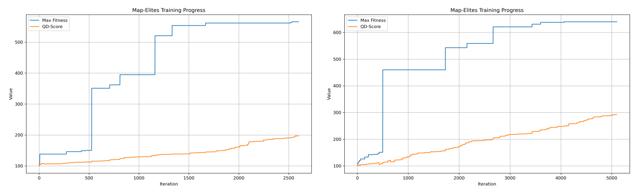

Figure 4: 2500 (left) and 5000 (right) max fitness and QD-score for behavior space of SNEW for map 1.

Comparing the max fitness of SNEW with 5000 iterations with the max fitness of CEA (see figure 5), it can be seen that SNEW improves rapidly at around 500 iterations. Comparatively, CEA sees a big spike around the 50th generation. At this point it will have run the simulation about 1000 times. After this, the improvements are more steady. CEA ends up finding a higher performing castle in this instance. However it should be noted that it runs many more simulations since it has as population of 20 and 500 iterations, meaning a total of 10000 simulations are run.

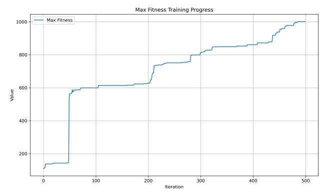

Figure 5: CEA max fitness for map 1.

5.2 Map 2

In the second map, the castle is situated at the side of a mountain, with a path running around it.

The comparison of coverage for SNEW and AC (figure 6) show largely similar results as for map 1. This indicates that the behavior space show the same characteristics even on different maps.

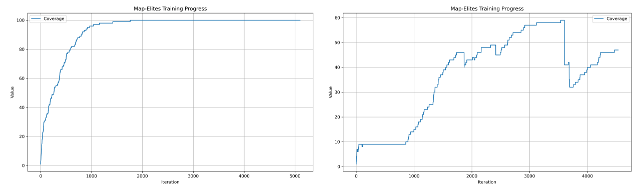

Figure 6: SNEW (left) and AC (right) coverage for a run with 5000 iterations for map 2.

Comparing the iteration size for map 2 will be done for behavior space AC, as SNEW was analyzed in the previous section. Looking at figure 7, it can be seen that the max fitness converges at about 250 after around 750 iterations and does not improve in subsequent iterations. The QD-score is also relatively flat.

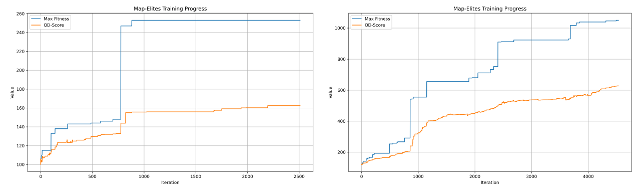

Figure 7: 2500 (left) and 5000 (right) max fitness and QD-score for behavior space AC for map 2.

Conversely, for 5000 iterations the QD-score is steadily improving, and could likely improve further with more iterations. One conclusion could be that 5000 iterations are just plain better than 2500 iterations. However, since the QD-score for 2500 looks so flat, another conclusion could be that the results of runs in behavior space AC are not that reliable. Further work be needed to establish this.

Comparing the results for behavior space SNEW with 5000 iterations with the results for CEA (see figure 8), it can be seen that the CEA algorithm finds a local maximum at around 350, and it doesn't break out of it. Meanwhile, the MAP-Elite run finds an individual performing 991, and holds an average of well above 400 for a full archive of 100 individuals.

The fact that CEA performs so much worse here shows that it can stuck in a bad local optimum.

5.3 Comparing Castles

To compare how the castles utilize the terrain, the two best castles from the MAP-Elites run with 5000 iterations in the SNEW behavior space are shown on figure 9.

In map 1, the castle places a lot of towers and two layers of walls so that the archers can shoot as many as the axemen

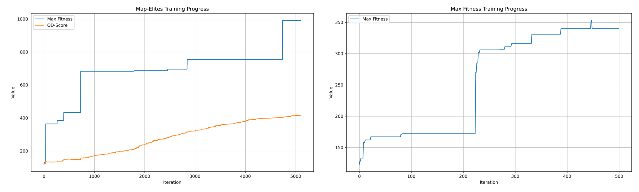

Figure 8: MAP-Elites with 5000 iterations and behavior space SNEW (left) and CEA (right) max fitness for map 2.

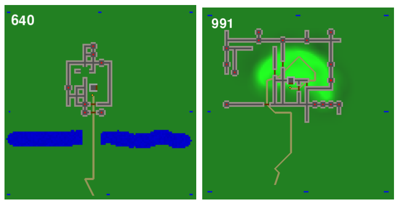

Figure 9: The castles with the highest fitness in SNEW behaviors space with 5000 iterations for map 1 (left) and map 2 (right).

as possible before they enter. Further, towers the lake works as a funnel, so that all axemen from the bottom will path find through the narrow pass.

In map 2, the structure is much wider. It narrowly avoids intersecting with the path to the south, creating gates inside the castle, and instead has a layered approach with archer towers all around. It makes sure that the attackers have to climb the mountain side, which slows them down.

We see that two very different structures emerge from the two different maps, as different maps require different solutions.

6 Discussion

Castles come in many shapes and sizes, and therefore we hypothesized that an illumination algorithm would perform better than a conventional evolutionary algorithm at adapting to the terrain and defending well. According to Mouret and Clune, in terms of illumination algorithms, MAP-Elites generally find better solutions than i.e. Novelty Search + Local Competition (Mouret and Clune 2015).

Our results show that although CEA can find good solutions as seen on map 1, the results from map 2 indicate that it is less reliable if it gets stuck in a local optimum.

Our longest runs of MAP-Elites run for 5000 iterations. Such a run takes about 3 hours to run on our benchmark system. As seen in the results, the QD-score seems to be still be improving. More iterations likely would have lead to a better QD-score, and more iterations would be achievable with *faster* iterations. We identified that the performance bottleneck of the algorithm was the simulation, specifically the path-finding. Attempts were made to multi-thread the path-finding. However, Pythons Global Interpreter Lock (GIL) meant that no speedup was achieved (Python Wiki 2020). Further work could port the path-finding code to C or C++ and then call that code from Python and release the interpreter lock.

One challenge was handling when the castles were trying to generate ”out-of-bounds”, meaning that the instructions would try to build over the *keep*, or outside the buildable area. One approach could be to simply discard any castle tries to build ”out-of-bounds”. Another approach could have been to implement a Constrained MAP-Elites, keeping a feasible and infeasible set of solutions for each cell (Gravina et al. 2019). Implementing the self-sensitive aspect of the L-systems developed by Parish and Müller could also have been pursued (Parish and Müller 2001). Our solution was to simply not build anything out of bounds, and keep the agent at the edge of the bounds until it turned and starting building another direction. While this worked, it may have been part of some issues we had with *locality*.

As mentioned in section 2.3, the a small change to a genotype with high locality should lead to a small change in the phenotype. In our grammar, if some **LEFT** or **RIGHT** instruction was removed or added, the entire castle thereafter would be rotated 90 degrees. This results in a very volatile genotype to phenotype mapping. Mouret and Clune find that crossovers that lead to new elites are often found by sampling nearby individuals in behavior space (Mouret and Clune 2015). However, due to low locality, this is not necessarily the case for our genotype.

Quite a large simplification of our battle simulation is that the attackers only can break down gates. In the work by Mas et al., their combat simulation includes siege towers, catapults and a wall breaking system to allow any wall to be breached. Including something like this could have lead to more interesting castles that were stronger on all sides.

Further tweaks could be made to the fitness functions and other parameters, such as unit speed, health, archer damage etc. For instance, the fitness function that only rewards simulation steps would reward a castle that kills all attackers quickly with a worse score than a castle that holds out for a long time, yet is defeated eventually. Giving a large reward for defeating all attackers might result in a more comparable fitness assessment.

7 Conclusion

The project shows that when using MAP-elites for building castles and running attack/defense based simulations on inherently different maps, the constructed castle will differ in nature in accordance to the terrain. Additionally, since the environment allows for a variation of good solutions, MAP-Elites tends to find multiple different designs with high fitness, where as the narrower search space of the conventional genetic algorithm has it finding only one variety of castle for each run. It also shows that in this case MAP-Elites tends to find castles with a higher fitness score sooner and more consistently than a conventional genetic algorithm. However although the margin of failure of the conventional genetic algorithm can at least in part be attributed to the high volatility of the genotype.

Drawing from Dusterwald’s work, investigating how to place the castles in such a way that it is not placed on water or on steep mountain side is a topic from further work (Dusterwald 2015). Evolving the attacking side as well as the defending side could be achieved by incorporating Generational Adversarial MAP-Elites (Anne et al. 2025), and could result in more castles adapting to better adversaries over time. Adapting a less volatile grammar, such as a shape grammar (Müller et al. 2006), could also be a direction for further work.

References

Anne, T.; Syrkis, N.; Elhosni, M.; Turati, F.; Legendre, F.; Jaquier, A.; and Risi, S. 2025. Adversarial Coevolutionary Illumination with Generational Adversarial MAP-Elites. arXiv:2505.06617 [cs].

Dusterwald, S. 2015. *Procedural Generation of Voxel Worlds with Castles*. PhD Thesis, University of Waikato.

Fontaine, M. C.; Lee, S.; Soros, L. B.; De Mesentier Silva, F.; Togelius, J.; and Hoover, A. K. 2019. Mapping hearthstone deck spaces through MAP-elites with sliding boundaries. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 161–169. Prague Czech Republic: ACM.

Gravina, D.; Khalifa, A.; Liapis, A.; Togelius, J.; and Yannakakis, G. N. 2019. Procedural content generation through quality diversity. In *2019 IEEE Conference on Games (CoG)*, 1–8. IEEE.

Interregion.cz. 2018. Castle Trosky.

Konami. 1986. Castlevania.

Mas, A.; Martin, I.; and Patow, G. 2020. Simulating the Evolution of Ancient Fortified Cities. *Computer Graphics Forum* 39(1):650–671.

Mitchell, M. 1998. *An introduction to genetic algorithms*. MIT press.

Mojang Studios. 2011. Minecraft.

Mouret, J.-B., and Clune, J. 2015. Illuminating search spaces by mapping elites. arXiv:1504.04909 [cs].

Müller, P.; Wonka, P.; Haegler, S.; Ulmer, A.; and Van Gool, L. 2006. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers on - SIGGRAPH ’06*, 614. Boston, Massachusetts: ACM Press.

Nintendo EAD. 1996. Super Mario 64.

Parish, Y. I. H., and Müller, P. 2001. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 301–308. ACM.

Python Wiki. 2020. GlobalInterpreterLock - Python Wiki.

Shaker, N.; Togelius, J.; and Nelson, M. J. 2016. *Procedural Content Generation in Games*. Computational Synthesis and Creative Systems. Cham: Springer.

TaleWorlds Entertainment. 2022. Mount & Blade II: Bannerlord.

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. Publisher: IEEE.

Warhorse Studios. 2025. Kingdom Come: Deliverance II.