Designing a Template-Based Map Generator for Heroes of Might & Magic III

(Stworzenie generatora map do Heroes of Might & Magic III opartego o szablony)

Dawid Skowronek Grzegorz Kodrzycki

Praca inżynierska

Promotorzy: dr Jakub Kowalski

mgr inż. Radosław Miernik

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

2 lutego 2025

Abstract

This work presents a template-based map generator for Heroes of Might & Magic III (HOMM3) and is inspired by the challenges explored in prior research [1]. The generator creates maps compatible with the commonly used open-source recreation of the HOMM3 engine – VCMI [2]. In this work, we introduce our method of creating maps and explain how to use our generator.

Our approach uses templates to allow creating maps even by inexperienced players. We provided detailed descriptions of the algorithms and heuristics used to place objects, resources, towns, guards and other elements. Additionally, we discuss the implementation of our generator and provide an example map.

The generator's maps are compatible with VCMI, which allows for extensive testing, ensuring that maps meet desired criteria for balance and playability. We also highlight potential improvements and future work, such as incorporating underground levels and water travel, to further enhance the map generation process.

This work aims to contribute to the HOMM3 community by providing a robust tool for creating high-quality maps, enhancing the overall experience.

Streszczenie

W niniejszej pracy przedstawimy bazujący na szablonach generator map dla Heroes of Might & Magic III (HOMM3), zainspirowany wyzwaniami przedstawionymi w pracy [1]. Generator tworzy mapy kompatybilne z popularną, otwartoźródłową rekreacją silnika HOMM3 – VCMI [2]. W tej pracy prezentujemy naszą metodę tworzenia map oraz wyjaśniamy, jak korzystać z naszego generatora.

Nasze podejście korzysta z szablonów, aby umożliwić tworzenie map nawet niedoświadczonym graczom. Szczegółowo opisujemy algorytmy i heurystyki użyte do rozmieszczenia obiektów, zasobów, miast, strażników oraz pozostałych elementów. Ponadto omawiamy implementację oraz przedstawiamy przykład wygenerowanej mapy.

Wygenerowane mapy są kompatybilne z VCMI, co pozwala na dokładne testy, zapewniając, że spełniają one kryteria sprawiedliwości oraz grywalności. Wskazujemy również możliwe kierunki rozwoju naszej pracy, takie jak dodanie możliwości tworzenia podziemi oraz akwenów wodnych.

Celem tej pracy jest wsparcie społeczności HOMM3, dając użytkownikom do dyspozycji narzędzie do tworzenia dopracowanych map.

Contents

1 Introduction **7**

2 Heroes of Might and Magic 3 **9**

2.1 Tools 11

2.2 Key concepts 12

2.2.1 Zones 13

2.2.2 Terrains 13

2.2.3 Factions 15

2.2.4 Towns 15

2.2.5 Mines 16

2.2.6 Special buildings 17

2.2.7 Guards 17

2.2.8 Obstacles 18

2.2.9 Collectibles 19

3 Template Description **21**

3.1 General map information 21

3.2 Zone information 23

3.3 Connection information 24

3.4 Example template 25

4 Map Generation **27**

4.1 Zone generation 27

4.2 Town placement 29

4.3 Border and connection of zones generation 29
4.3.1 Determining zone borders 30
4.3.2 Finding connection points 31
4.3.3 Setting wide connections 31
4.4 Object placement 32
4.4.1 Mines placement 32
4.4.2 Treasures placement 33
4.5 Road placement 35
4.6 Guard placement 36
4.6.1 Guard attributes 36
4.7 Noise placement 38

5 Fairness of the Map **41**
5.1 Improvements in map generation 41
5.2 Other ideas to test fairness 41

6 Conclusions and Future Work **43**

Bibliography **45**

Chapter 1

Introduction

Heroes of Might & Magic III [3], released in 1999, remains one of the most iconic turn-based strategy games. Central to its enduring appeal is the ability to create and explore diverse maps. However, manually designing engaging maps can be time-consuming and challenging, particularly when maintaining fairness and consistency.

One of the key factors in its continued popularity is the community-driven project VCMI [2], which tries to recreate the original HOMM3 engine. As it provides the ability to extend the base game, it is a commonly used alternative by many players. Therefore, we decided to generate maps that can be played on this engine.

Another key factor in the game's enduring popularity is the large number of maps created by fans and published online for other players to download and review [4]. These maps introduce new challenges and strategic elements, making gameplay more diverse and engaging, allowing users to try something else besides the base maps available in the original client. However, this creates a new issue: hand-crafted maps are time consuming and often difficult to balance. To address this, many players and game designers use algorithms to generate maps. This speeds up the process, but often results in less engaging designs. We decided to tackle this problem and will present our approach to this process.

There are several ways to approach this problem. For example, in the random map generator used in Songs of Conquest (SoC) [5], a game often regarded as a successor of the HOMM series, the templates are divided into two separate parts: *layout* and *blueprint*. The layout is responsible for describing the region layout, connections, and overall appearance of the map. The blueprint focuses on object placement, their quantity, and modifiers, such as whether they are guarded. However, in our work, we decided that combining these functionalities into a single template suits us best.

We also observed a lack of comprehensive research on this topic and available studies often provide limited explanations of the creation process. In our work, we thoroughly detail the structure of the templates, enabling anyone to use our generator. Additionally, we outlined the map creation process step by step, making it easier for future users to understand and build upon.

Chapter 2

Heroes of Might and Magic 3

In HOMM3, players take turns exploring the map, collecting resources, capturing mines, managing towns, and commanding armies led by heroes. Usually, the objective is to defeat all opponents. Other possible objectives are to achieve specific victory conditions, such as capturing key towns or gathering unique artifacts. Figure 2.1 shows how the game looks in the HOMM3 GUI. All interface elements are described below and referenced using numbers (e.g., ①).

This GUI consists of two main sections: *playable elements* and *menu/navigation*. Here is a brief overview of each, but we will explore them in detail later.

Playable elements are interactive components on the map where players make strategic decisions and directly influence the game world. Towns ① are central structures where players recruit units, build buildings, and manage their growth. Another important element is heroes ② – characters controlled by the player that explore the map, collect resources, and engage in battles. Buildings that produce resources, commonly referred to as mines, are the backbone of every player's economy. These structures generate a consistent supply of a specific resource at the beginning of each day. For example, the sawmill ③ produces wood.

But mines are not the only way to boost the economy. A good example are piles of resources ④, (such as wood and ore in the picture), which the player's hero can collect once. In addition to towns and mines, we also have special buildings ⑤ that grant unique effects depending on the type of structure. Each hero can equip artifacts ⑥ to gain buffs or special abilities. These can be found in a specific location or collected during exploration. Other important collectibles are treasure chests ⑦ that offer a choice between gold and experience, but sometimes can also grant artifacts or resources. Lastly, map designers place roads ⑧ to guide players and give them movement speed.

We also have GUI elements that provide important information and shortcuts to streamline gameplay. For example, the resource bar ⑨ which displays the current stockpile of resources, the heroes list ⑫, and the towns list ⑬.

When we choose hero, we can see additional information in the hero panel (10), where we can see stats and army count info. In addition to this game information, we also have some buttons (14) to perform essential actions, such as entering a town, ending the turn, or switching between the surface and the underground map layers.

Figure 2.1: HOMM3 GUI.

2.1 Tools

The foundation of our work was based on `homm3tools` [6] and `homm3lua` [7]. These tools enabled precise object placement and map modification with a tile-level accuracy. The generated maps were validated using the VCMI engine.

`homm3tools` is a comprehensive project that provides a suite of tools and libraries for HOMM3. It is designed to facilitate in-game modifications, making it an essential resource for custom map creation.

`homm3lua` serves as a Lua API for `homm3tools`. Its integration with Lua scripting empowers users to create custom scripts to modify gameplay elements, significantly enhancing the overall experience of working with `homm3tools`.

VCMI is an open-source community project aimed at recreating and extending the original HOMM3 engine. Beyond enabling users to play the game, it offers utilities like *vcmieditor*, which allows for easy inspection of map designs, and *vcmilauncher*, which supports running bots on custom maps. These tools make VCMI a powerful resource for creating and testing generated maps.

2.2 Key concepts

Figure 2.2: Example of a fully generated map.

2.2.1 Zones

A map in HOMM3 can be conceptualized as a collection of zones. Zones are distinct areas on the map, each with terrain type, resources, and guards. They define the layout of the map, influencing movement and exploration.

Zones can serve different purposes: some may contain more resources and mines to boost the player's economy, while others may be designed as combat zones where players can gain more experience. Typically, starting zones should strike a balance between resource richness and difficulty.

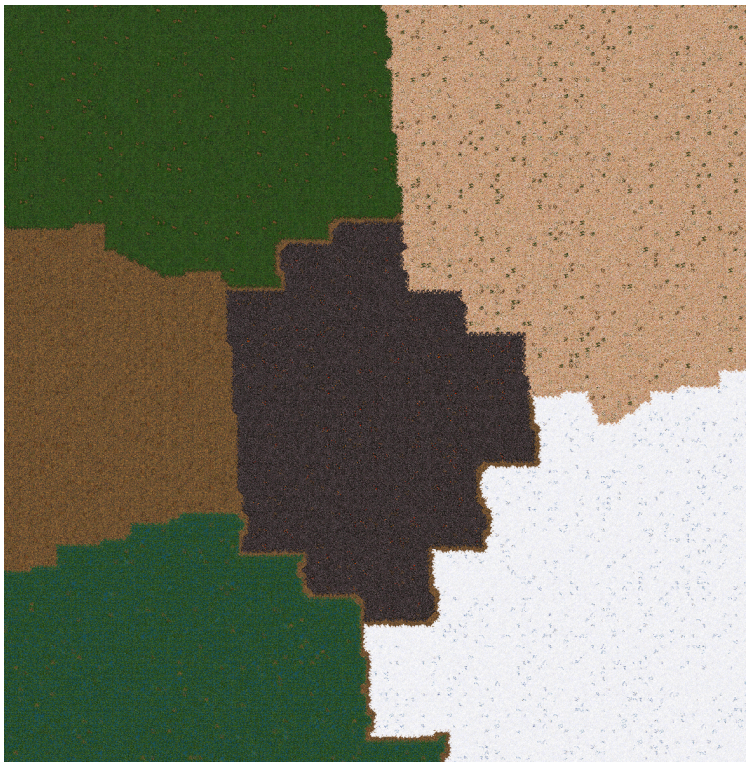

Figure 2.3: Example of zones without any key components.

2.2.2 Terrains

Terrain serves multiple purposes, from aesthetic differentiation of zones and enhancing the map's visual appeal to impacting hero movement and granting extra stats to creatures during combat. It can be divided into two types: *basic* and *magical*.

Basic type include: *grass*, *sand*, *snow*, *swamp*, *rough*, *subterranean*, *lava*, *water*, *dirt* and *rock*, while magical types include *magic plains*, *cursed ground*, *clover fields*, *rockland*, *ground* and several others.

Basic ones only impact movement speed, with the exception of water, which requires a special artifact, spell, or ship to travel. For example, traveling through rough terrain increases movement cost by 125%, while on grass it remains unaffected.

Each faction has its *native terrain*, which provides extra attack, defense, and speed for creatures fighting on their native terrain.

Magical terrain, on the other hand, can overlay the basic one. If a battle occurs on these lands, they have a significant impact on the game. For example, the magic plains cause all spells to be cast at expert level, while the holy ground gives all good-aligned creatures +1 morale, and all evil-aligned ones -1 morale [13].

Although the tools we use allow for the placement of magical terrains, we did not include this option in our template for this work.

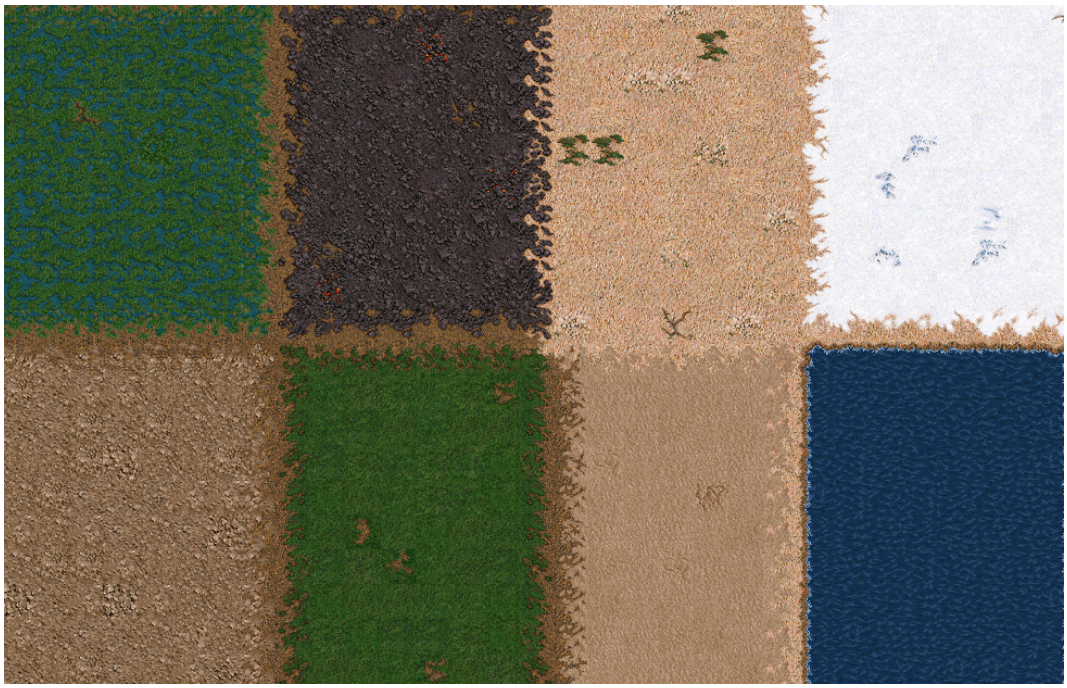

Figure 2.4: Example basic terrains. From top-left: swamp, lava, sand, snow, rough, grass, subterranean and water.

2.2.3 Factions

A faction is a distinct group with its own unique town, units, abilities, and playstyle. Each of them offers a different experience and requires specific strategies. Factions are central to the game's diversity as they require players to adapt their strategies accordingly. In our generator, we support all factions from the basic version of the game: (Good Factions) *Castle*, *Rampart*, *Tower*, (Evil factions) *Inferno*, *Necropolis*, *Dungeon*, (Neutral factions) *Stronghold*, *Fortress*.

2.2.4 Towns

Towns are the cornerstone of the player strategy. They are the main places where players recruit troops, build structures, and generate income. Typically, each player starts with at least one town. Every faction has unique towns, each with its own specialties, needs, troops, and upgrades. Some towns may require more gold, while others are more resource-based. Controlling towns is crucial for maintaining a steady flow of resources and troops, making them strategic points on the map.

Figure 2.5: All possible towns from basic version of the game.

2.2.5 Mines

Mines are resource-generating structures that provide specific types of resources. We allow for placing the following mines:

- *Sulfur dune* for sulfur,
- *Alchemists lab* for mercury,
- *Ore pit* for ore,
- *Crystal cavern* for crystal,
- *Gem pond* for gems,
- *Gold mine* for gold and
- *Sawmill* for wood.

Controlling mines ensures a steady supply of materials needed for developing towns and recruiting troops. They are particularly valuable in early- and mid-game stages.

Figure 2.6: All possible mines. From top-left: Sulfur dune, Alchemists lab, Ore pit, Crystal cavern, Gem pond, Gold mine and Sawmill.

2.2.6 Special buildings

Special buildings are unique structures found across the map that offer various benefits, such as buffs to your heroes or troops. These buildings may significantly enhance player's capabilities and provide a competitive edge. They may also grant the ability to buy artifacts or gain knowledge, which may be crucial, especially in mid- and late-game.

Figure 2.7: Example special buildings.

2.2.7 Guards

Guards are neutral creatures stationed at specific locations on the map to protect valuable resources, mines, or access to other zones. Defeating guards is a key part of map progression, as it allows players to claim resources and expand their influence. The strength and type of guards vary based on the map design and difficulty level.

However, the level of guards alone is not sufficient to determine their difficulty. For example, ranged or high-mobility units are more likely to cause losses, as they can attack from a distance or quickly close the gap with the player's units. In contrast, slow melee creatures often can be *kited* (outmaneuvered using fast/cheap units while damaging it with ranged ones) or killed before they even get close, allowing players to avoid taking any damage while defeating them.

On top of that, there are also unit-specific abilities, which can also impact the battle result. For example, the Vampire Lord's lifesteal ability allows it to regain health by dealing damage, which can prolong fights and possibly increase losses. Similarly, the Crusader's ability to strike twice significantly increases its damage output. Therefore, the composition and abilities of the guards play a crucial role in assessing the challenges they present.

Figure 2.8: Example creatures from (top to bottom rows) Castle, Rampart, Tower and Inferno.

2.2.8 Obstacles

Obstacles are terrain features or structures that restrict movement and access to certain areas. They can be neutral, like mountains, rivers, or trees, or controlled, like gates. Obstacles are a key part of enforcing exploration and determining paths.

Figure 2.9: Example obstacles.

2.2.9 Collectibles

Collectibles include items like treasure chests, resources, and artifacts scattered across the map. These items provide immediate benefits, such as one-time resources or experience gains or granting powerful items like artifacts. Collecting these items can significantly boost a player's progress, especially in the short term.

Figure 2.10: Example collectibles. The top row contains resources and a treasure chest, while the other rows include example artifacts.

When designing a balanced and engaging map, it is crucial to carefully consider the difficulty of each zone, its *richness* (a parameter indicating the number of collectibles within a zone), and how these elements encourage players to explore the map more thoroughly. Striking the right balance ensures fairness while maintaining the strategic depth and enjoyment of gameplay.

Chapter 3

Template Description

In this chapter, we will describe the structure and components of the templates used in our generator. These templates are written in JSON format, allowing for easy readability and modification. We will break down the template into parts and explain each section. At the end, we will present a fully functional example.

3.1 General map information

```
{
    "name": "2P Duel Template",
    "description": "Map for 2 players with a contested central area.",
    "size": "S",
    "difficulty": "Easy"
}
```

Listing 1: General information about the map.

The `name` and `description` fields provide information that is visible to players when they search for the map in the main menu. These fields should be straightforward and easy for the player to understand.

The `difficulty` field is purely cosmetic; we do not use this parameter when generating a map. It appears in the main menu to inform players about the map author's perceived difficulty level.

The `size` parameter significantly impacts the map, as it determines the number of tiles. Possible sizes are `S/M/L/XL`, with `S` being 36×36 tiles up to `XL` being 144×144 tiles.

Figure 3.1 shows how this information will be displayed in the map menu.

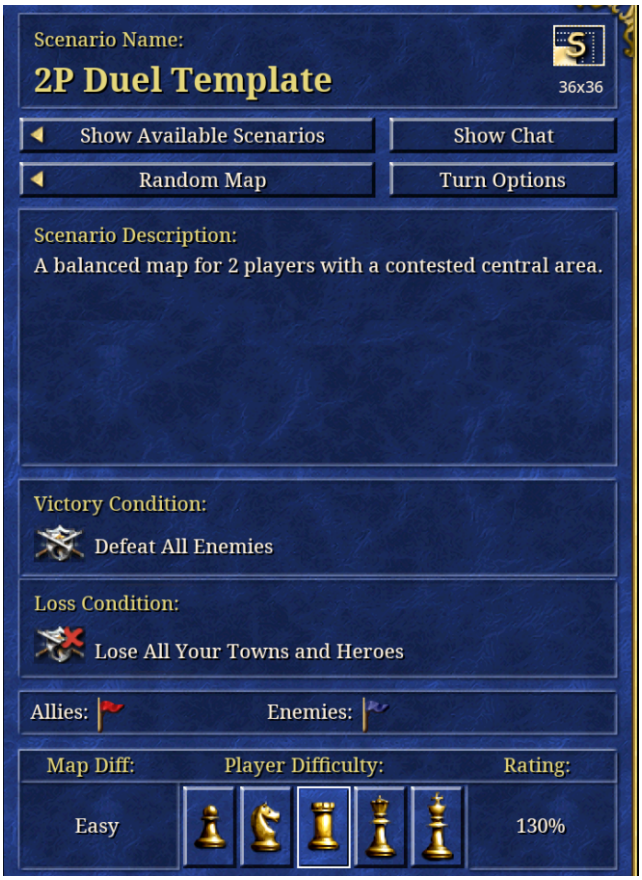

Figure 3.1: Map menu selection.

3.2 Zone information

We have created abstractions to enable the individual definition of parameters for each zone. In this section, we will go through each field and explain its purpose in the process of creating the map.

```
{
      "id": 1,
      "size": "M",
      "terrain": "Grass",
      "richness" : "Low",
      "difficulty": "Beginner",
      "numberOfTowns": 1,
      "towns" : [
        {
          "faction" : "Stronghold",
          "owner" : "Player_1"
        }
      ],
      "maxNumberOfMines": 3,
      "numberOfMineTypes": 1,
      "mines": [
        {
          "type": "Gold Mine",
          "owner": "Player_1",
          "minCount": 2
        }
      ]
}
```

Listing 2: General information about zone.

The `id` of the zone is an abstraction used mainly to identify which zones are connected.

The `size` of the zone determines how large it will be. Larger zones may contain more treasures and mines, but players may need more time to explore them. Possible values are `S/M/L/XL`; they will inform the generator how big a terrain portion this zone should have. This attribute is not correlated with size of the map.

The `terrain` of the zone can be chosen by the creator, even if some factions do not benefit from specific terrains. Additionally, there is the possibility of selecting a random terrain type. We believe that this flexibility allows for more interesting map designs. Possible values are `Grass`, `Sand`, `Snow`, `Swamp`, `Rough`, `Subterranean`, `Lava`, `Dirt`, `Rock`, and `Random`.

The `richness` field is responsible for manipulating the amount of treasures

and buildings in the zone. Finding the right parameter is crucial to creating a fair economy for every player. Possible values are `Low`/`Medium`/`High`.

The `difficulty` field indicates how challenging a zone should be by type and quantity of guards in the zone. Guards are placed at connections and near crucial objects. Possible values are `Beginner`, `Easy`, `Normal`, `Hard`, `Expert`, and `Impossible`.

The array of `towns` describes the towns within the zone, including their faction and ownership. As mentioned above, the faction influences gameplay significantly for players. We also provide an option to make it neutral, by setting the ownership of a town to `"None"`. Unlike some map generators, our version does not support random town generation.

The array of `mines` specifies the types of mines in the zone. Users can also define ownership and specify the minimum number of each mine type they want to see. Additionally, the `maxNumberOfMines` field sets the maximum number of mines in the zone.

3.3 Connection information

```
{
      "zoneA": 1,
      "zoneB": 6,
      "type": "monolith",
      "tier": 2
}
```

Listing 3: General information about a connection.

In this section, we store information about the connection between zones. The `zoneA` and `zoneB` fields hold the IDs of the zones we want to connect.

The `type` of connection specifies how the zones are linked, with three options currently available: `narrow`, `wide`, and `monolith` (`portal`). We will describe them in detail later.

The `tier` of the connection indicates the level of the road used, affecting the movement speed of players. Higher-tier roads allow heroes to move faster. Possible values for road tiers are `0`/`1`/`2`/`3`. A tier of `0` indicates no road, meaning movement costs are unchanged. For higher tiers, the movement cost is progressively reduced: tier `1` to 75%, tier `2` to 65%, and tier `3` to 50%. Additionally, roads at each tier have distinct visual appearances, making it easy to differentiate their quality and associated movement efficiency.

3.4 Example template

```
{
  "name": "2P Duel Template",
  "description": "Balanced map for 2
  ↪  players.",
  "size": "S",
  "difficulty": "Easy",
  "zones": [
    {
      "id": 1,
      "size": "M",
      "terrain": "Grass",
      "richness": "Low",
      "difficulty": "Easy",
      "numberOfTowns": 1,
      "towns": [
        {
          "faction": "Stronghold",
          "owner": "Player_1"
        }
      ],
      "maxNumberOfMines": 4,
      "numberOfMineTypes": 2,
      "mines": [
        {
          "type": "Ore Pit",
          "owner": "Player_1"
        },
        {
          "type": "Sawmill",
          "owner": "Player_1"
        }
      ]
    },
    {
      "id": 2,
      "size": "M",
      "terrain": "Random",
      "richness": "Low",
      "difficulty": "Easy",
      "numberOfTowns": 1,
      "towns": [
        {
          "faction": "Inferno",
          "owner": "Player_2"
        }
      ],
      "maxNumberOfMines": 4,
      "numberOfMineTypes": 2,
      "mines": [
        {
          "type": "Ore Pit",
          "owner": "Player_2"
        },
        {
          "type": "Sawmill",
          "owner": "Player_2"
        }
      ]
    },
    {
      "id": 3,
      "size": "S",
      "terrain": "Sand",
      "richness": "High",
      "difficulty": "Normal"
    }
  ],
  "connections": [
    {
      "zoneA": 1,
      "zoneB": 3,
      "type": "narrow",
      "tier": 2
    },
    {
      "zoneA": 2,
      "zoneB": 3,
      "tier": 2
    }
  ]
}
```

Chapter 4

Map Generation

Map generation is quite a complex task to tackle. Dividing it into smaller parts allows for code modularization, making the work a bit easier. Each step uses the same random seed to ensure consistency.

4.1 Zone generation

In this step, we aim to generate zones with terrains while preserving connections defined in the template as much as possible. Our main heuristic here is to maximize the distances between the zones if there is a connection between them.

First, we need to find N such $N^2 \geq \#Zones$. Then, we create an $N \times N$ grid. Starting from the first zone, we place it on a random edge of the grid. Then, we place zones in each cell while maximizing our heuristic.

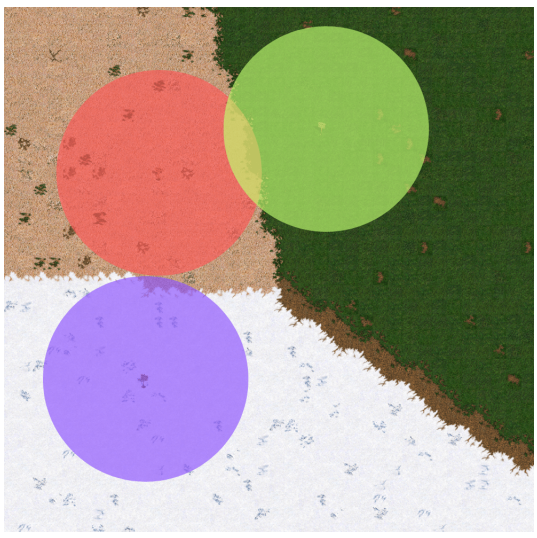

Figure 4.1: Zones placed within the grid.

After initializing the grid, we divide our map and randomly assign each zone's center within cells. The next step is to use the Fruchterman-Reingold algorithm [8] to find better centers for the zones. This algorithm assumes that we try to fit N circular zones with a radius on a map. The idea of this algorithm is that connected zones attract while intersecting zones push back.

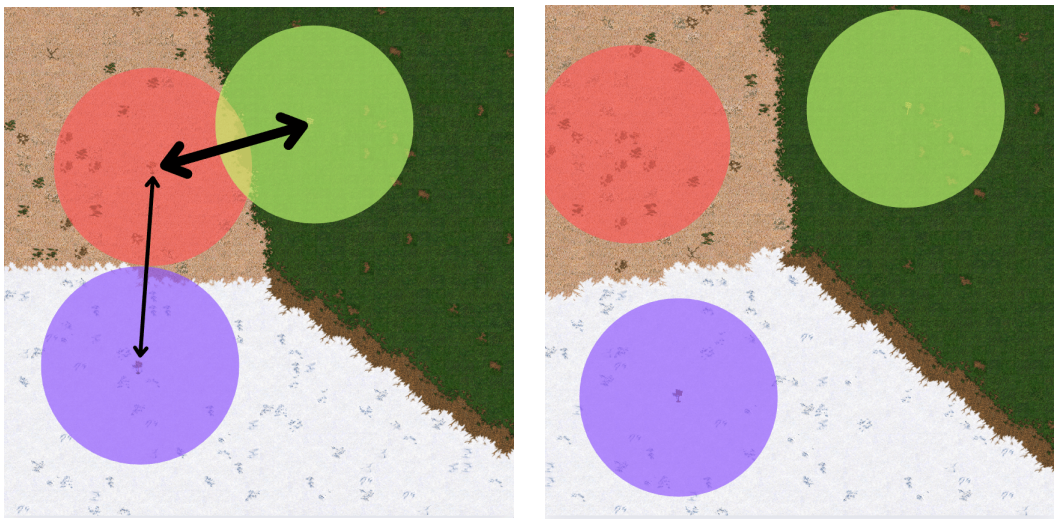

Figure 4.2: Fruchterman-Reingold visualization. The thicker arrow between the green and red zone represents a stronger force separating them.

The final step is to assign the corresponding terrain type to each tile. In the first version, we assigned terrain from the closest zone. Another approach is to generate Penrose Tiling [9] and assign vertices from this tiling to the nearest zone (therefore, the desired terrain type), then assign each pixel to the closest vertex.

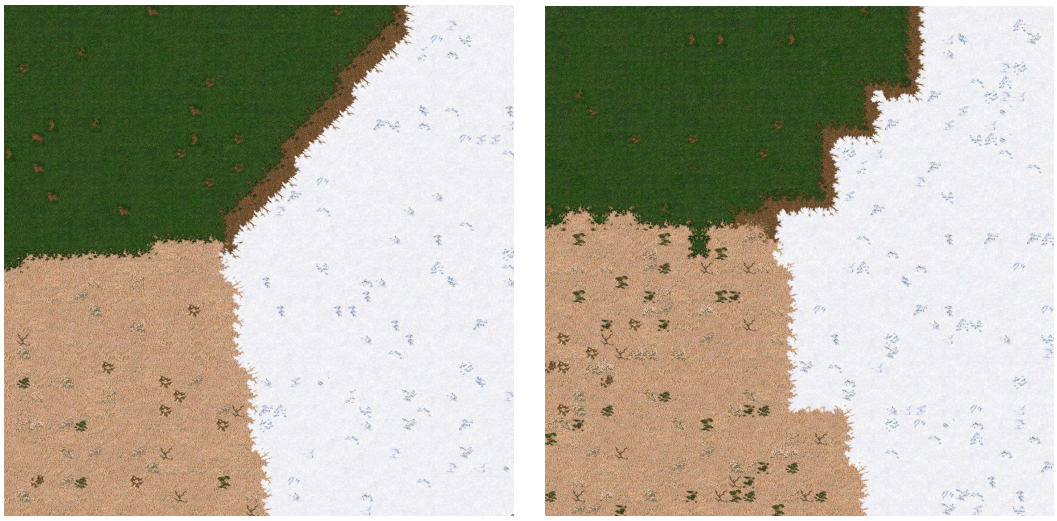

Figure 4.3: Comparison between map with and without Penrose Tiling.

Other generators have solved this problem using different methods. For example, in [10], the main idea is to use Voronoi diagrams to create zones. Then, they introduce irregularity using fractal randomization on zone edges.

4.2 Town placement

In this section, we focus on placing towns. The position for each of them is set to the center of mass of its respective zone. We believe that, after applying previous algorithms, placing towns in such locations brings fairness to the map. It is crucial to place towns early in the process, as subsequent steps heavily depend on towns' positions.

Figure 4.4: Map after towns placement.

4.3 Border and connection of zones generation

Generating borders and finding connections is done in a few steps:

1. Determining zone borders
2. Finding connection points
3. Setting wide connections

We allow for three types of connections:

- `Narrow` - zones are almost completely cut off, with only one guarded path.
- `Monolith` - zones are connected via monoliths (portals)
- `Wide` - there are almost no obstacles on the borders and border is not guarded in any way

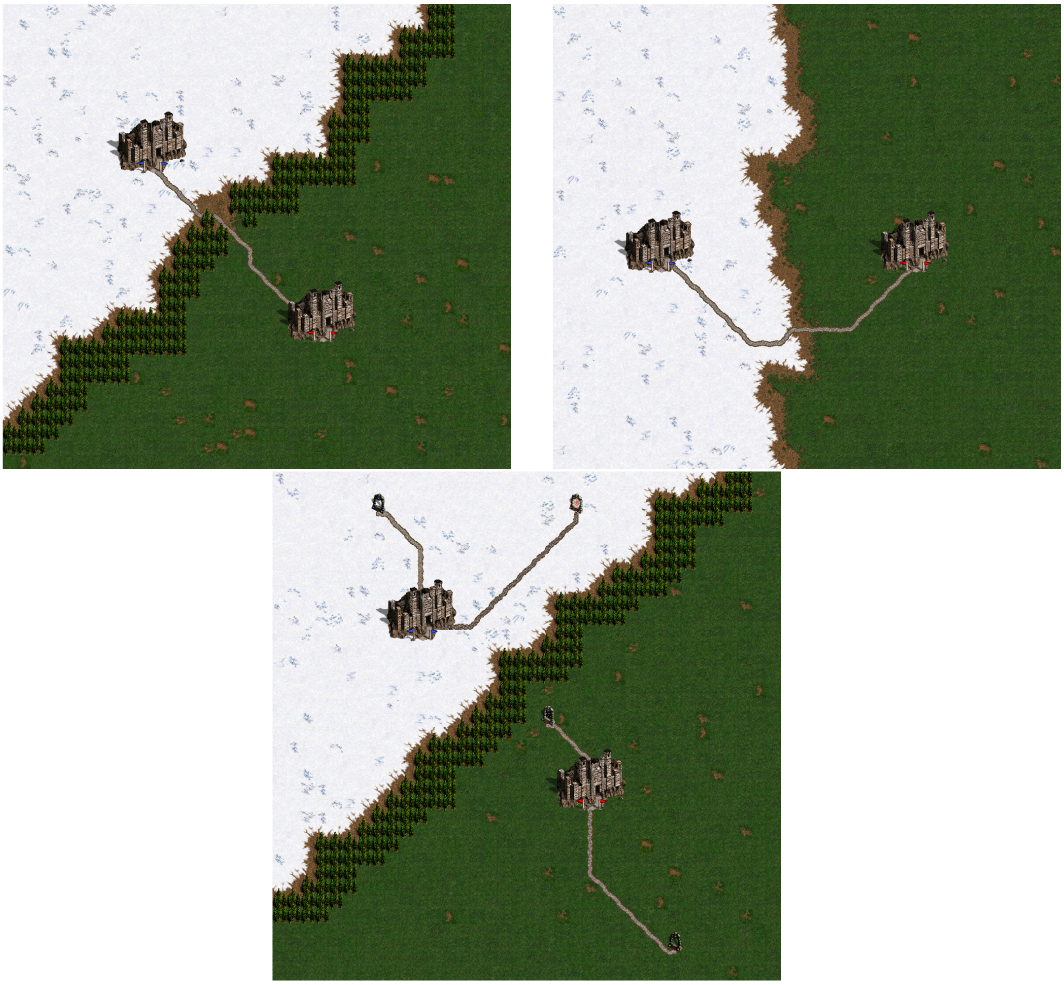

Figure 4.5: Comparison between Narrow, Wide and Monolith connections.

4.3.1 Determining zone borders

In this step, we iterate over the tiles and mark a tile as *border tile* if it is adjacent to a tile from another zone, considering all eight neighboring tiles in an octile connectivity pattern.

4.3.2 Finding connection points

After creating borders, we choose *connection points* for each zone we want to connect:

- Town with a monolith entrance
- Two points next to the zones' edges, which determine the starting points of roads for each zone (two points that belong to different zones and are not border tiles)
- Town with the starting point of a road between zones described above

If there is no town in the zone and we are trying to make one of the first two connection types, then we use the zone center instead.

We find those points using Dijkstra's algorithm. We run it from the town (or the zone center), returning the last point from the starting zone that is not a border and the first point from the other zone that is also not a border. Importantly, we do not place roads right away, we first place other objects on the map to avoid restricting tiles.

If zones are not directly connected after generation and a narrow connection is not possible, we consider the zones to be connected via monoliths and proceed with the algorithm.

When placing monoliths, we select a random valid tile in the given zone. A tile is considered *valid* if it is not a border, road, or gate, and there is at least one free tile between the chosen one and the other objects.

4.3.3 Setting wide connections

In the first two steps, we set first two types of connections. If the connection type should be **wide** we need to remove found border tiles.

Figure 4.6: Map after borders placement.

4.4 Object placement

We classify mines, collectibles, and special buildings as *objects*. In this part, we begin with the placement of mines, followed by collectibles and special buildings.

4.4.1 Mines placement

Essential mines

First, we check if the template includes basic mines (Ore Pit, Sawmill). If so, we want to place one of each type near the town, within a distance that guarantees getting those mines in three turns.

Placing the minimum count of specified mines

Next, we place the mines specified in the template until there is at least the minimum count of each type in the zone. The positions of these mines are randomly distributed throughout the zone.

Placing random mines to hit `maxNumberOfMines`

To reach `maxNumberOfMines`, we randomly select mines from the template and try to place them as far as possible from already placed ones.

In our implementation, possible mines for zones are heavily dependent on the author creating template. While allowing advanced users to manually place specific mines in zones is a viable option, we aim to make the generator more user-friendly. To achieve this, we could add a field to manipulate the wealth of the zone. Using information from this field, we could place faction-based mines in the zones.

Figure 4.7: Map after mines placement.

4.4.2 Treasures placement

Resources near mines

We aim to place resources near the entrances of mines with a certain probability to boost the player's economy. These resources are positioned strictly near the entrance to ensure that guards also secure them.

Treasure block

We place blocks of treasure, with their size randomly selected, containing various items such as resources or artifacts. These items are categorized into different tiers. Positions of such blocks are randomly picked from all possible placements.

Treasure buildings

Lastly, we place buildings that provide players with temporary boosts. The number of these is distinct from treasure blocks, as we think that they serve a different purpose. Their positions are also randomly selected from all possible locations.

An area for potential improvement is the better placement of those objects. In [10], Gus Smedstad suggests the idea of density, which we have partially implemented in the final step of placing mines.

Figure 4.8: Map after treasures placement.

4.5 Road placement

We start the process by fixing the border lines. We look for tiles that are adjacent to at least three border tiles in the NSWE directions. This is a purely cosmetic addition and does not significantly impact the gameplay. Then, we run Dijkstra's algorithm again, ensuring that we will step on free tiles. We may choose ones just next to objects, but this is not obligatory.

Once we generate the entire path, we unmark necessary tiles if they were marked as borders. Additionally, if the path is not from a town, we find a tile surrounded by obstacles on any axis and designate it as *guardian tile*, where we will place a guard later in the guard placement phase.

Figure 4.9: Map after roads placement.

4.6 Guard placement

In previous steps, we marked tiles that should have a guard on them. We consider two distinct types of guard:

- Gate guards
- Treasures/mines guards

Each type has a different difficulty scale, determining creatures' level, quantity, disposition, and whether they can flee or grow in numbers.

4.6.1 Guard attributes

- **Level**: indicates creature strength. Each creature has a basic and upgraded form. If upgraded, we would add 0.5 to its level.
- **Quantity**: number of creatures the player will face in battle. Higher difficulty levels result in higher quantities.
- **Disposition**: refers to the behavior of guards. Types include:
  - `COMPLIANT` - always joins the army
  - `FRIENDLY` - likely to join the army
  - `AGGRESSIVE` - may join the army
  - `HOSTILE` - unlikely to join the army
  - `SAVAGE` - never join the army
- **Can flee**: determines if guards can flee from battlefield. It can be set to `true` or `false`.
- **Can grow**: determines if guards will grow in number over time. It can be set to `true` or `false`.

Parameters such as level, quantity, disposition, can flee, and can grow are chosen based on the zone difficulty level and guard type. The guard level and quantity are randomly selected from a range corresponding to the zone level, while the other parameters are predetermined. For example, if the zone difficulty is set to Hard, then treasure guards' levels can vary from 3 to 5, while gate guards with the same difficulty will range from 5.5 to 6.5. The quantity for both will be randomly selected from a range of 60–120; disposition will be set to `HOSTILE`, can flee to `false`, and can grow to `true`.

For each tile, we determine the appropriate guard type and randomly assign the level and other parameters depending on this difficulty.

Figure 4.10: Map after guards placement.

4.7 Noise placement

We generate Perlin Noise [11] on an $N \times N$ grid. We are generating noise in the $[-1, 1]$ range, so to get the binary output, we are taking only values ≥ 0. The next step is to apply this noise to free tiles. Then, we mark tiles that must remain reachable (e.g., roads, the bottom line of buildings).

The next step is running Dijkstra's algorithm to calculate a minimal number of obstacles from each town to be in a specific tile using the following rules:

- **if the tile is free:** maintain the number from the previous tile,
- **if the tile is occupied:** treat it as an obstacle,
- **if the tile was free and now is occupied:** add weights after moving to it (1 for orthogonal movement, 2 for diagonal movement).

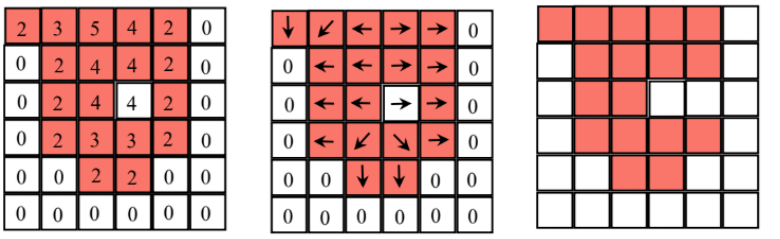

Figure 4.11: Visualization of algorithm for ensuring all areas are reachable [10].

After calculating these distances, we check all places that are supposed to remain reachable. If their distance $\neq 0$, we need to fix their reachability.

Figure 4.12: Final map after all steps.

Chapter 5

Fairness of the Map

Creating a fair yet asymmetrical map is quite challenging. During our design process, we developed heuristics with the goal of ensuring fairness in our maps.

5.1 Improvements in map generation

In [10], the concept of density is suggested as a heuristic to place objects. This approach involves storing additional information, *density*, for each object type, which specifies how many such buildings should exist within a zone. The first object of each type is randomly placed near our town (or the center of the zone). Other objects are placed as far as possible from each other.

Upon further consideration, we refined this heuristic to also keep core objects for players' economy at roughly equal distances from each player's town. While this introduces a bit of symmetry, it brings more fairness to the map.

5.2 Other ideas to test fairness

Previous improvements relied heavily on heuristics and our subjective perception of fairness. Fortunately, VCMI provides tools to play the specified maps with the prepared AIs. Using this opportunity, we created a simple script to run VCMI agents whose only task is to play games on our generated map. Then, we collect logs of games and create statistics for each player. To reduce the impact of game imbalance (some factions are stronger than others), we run games on maps with the same factions.

```
Enter number of games (default=3): 10
Enter number of batches (default=3): 10

Player Win Statistics:
Player 0 (red): 57 wins 57.00% Win ratio
Player 1 (blue): 43 wins 43.00% Win ratio
```

Listing 4: Statistics from an example map.

Unfortunately, this approach has one disadvantage: when `player_0` loses, the game ends. This means that for maps with more players, we are unable to gather all the necessary statistics. One solution we considered was to repeatedly swap all possible players with `player_0` to ensure that every player has the opportunity to be active until the end.

In the current version, we have implemented a script to test the map's fairness after generation. In future work, such information could be improved by using it during the generation process.

Chapter 6

Conclusions and Future Work

The main goal of this work was to develop a generator for fair maps based on a template for HOMM3 and to summarize the currently available tools to achieve this effect. We have thoroughly described the entire map creation process and presented the capabilities of our generator. All code related to this work is shared on GitHub [12]. Detailed instructions regarding the construction and usage can be found in the **Readme.md**.

We successfully created the generator, but we believe the topic is much more complex and that several other issues could be addressed.

HOMM3 allows for the creation of underground levels, which we omitted in our generator. Although this is a complex problem, we think it could significantly enhance the gameplay.

Additionally, the possibility of building ships and traveling by water allows for the creation of new zones and the introduction of a new type of connection. However, like the underground levels, this is a separate topic altogether.

We touched on maintaining map fairness, but there is room for a deeper analysis here as well. For example, we could more precisely analyze the locations where we place mines and other objects.

Bibliography

[1] J. Kowalski, R. Miernik, P. Pytlik, M. Pawlikowski, K. Piecuch and J. Sękowski, "Strategic Features and Terrain Generation for Balanced Heroes of Might and Magic III Maps," 2018 IEEE Conference on Computational Intelligence and Games (CIG)

[2] VCMI homepage
`http://vcmi.eu`

[3] Heroes of Might and Magic III homepage
`https://www.ubisoft.com/en-gb/game/heroes-of-might-and-magic-3-hd`

[4] Website with HOMM3 maps
`https://www.maps4heroes.com/heroes3/maps.php`

[5] Songs of Conquest "Random Map Generator Modding"
`https://www.songsofconquest.com/modding/rmg`

[6] `homm3tools` repository
`https://github.com/potmdehex/homm3tools`

[7] `homm3lua` repository
`https://github.com/radekmie/homm3lua`

[8] Fruchterman, Thomas MJ, and Edward M. Reingold. "Graph drawing by force-directed placement." Software: Practice and experience 21.11 (1991): 1129-1164.

[9] D'Andrea, F. 2023. A guide to Penrose Tilings.

[10] Gus Smedstad "The Heroes 3 Random Map Generator"
`https://www.dropbox.com/scl/fi/p6oadqz10bu4i24ieytba/heroes-3-random-map-generator-gus-smedstad.ppt?rlkey=v6rt1qcht4a0s205a9lvokvr7`

[11] Perlin, K. An image synthesizer. ACM Siggraph Comput. Graph. 1985, 19, 287–296

[12] Project's GitHub repo
`https://github.com/Sko0owi/HOMM3-mapgen`

[13] HOMM3 terrain wiki
https://heroes.thelazy.net/index.php/Terrain