Procedural Generation

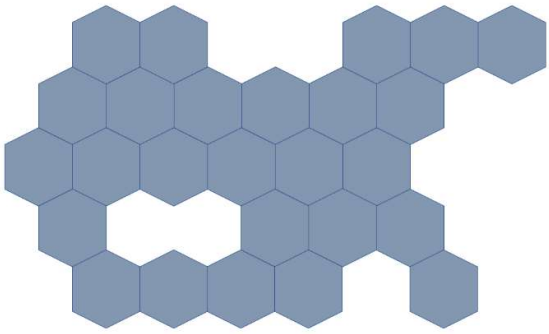
1 CONTEXT

To increase the replayability of our game, we want to have a different terrain for each match. The goal is to challenge the players and make them adapt to a new environment instead of memorizing an optimal strategy for a specific map they know everything about.

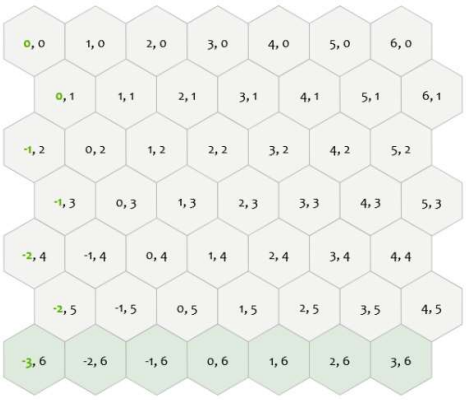
2 THE TERRAIN

The terrain is constructed using hexagonal tiles spread on a grid. The tiles content changes for every match.

We need to make a procedural generation algorithm for a partial grid. That would make it more modular and easily adaptable for other games or situations.



1 - A partial grid



2 - A full grid (a particular case of a partial grid)

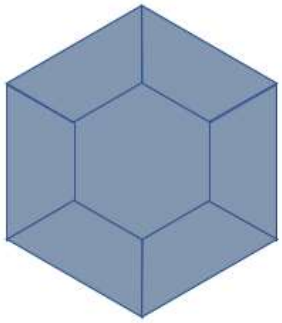
3 THE ALGORITHM

The algorithm will be a modified form of WFC (Wave Function Collapse). Inspired by the articles (1) and (2), I will add some features to have more control on the final generation.

WFC completes the grid tile by tile, using placement constraints to place each tile.

I will be using examples coming from our project. Most elements may be subject to change depending on our use case. The way the algorithm works, however, will be the same.

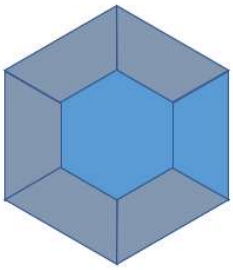
3.1 THE TILES



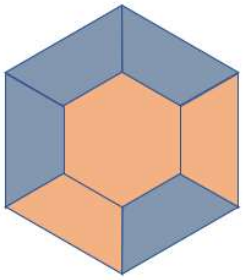
3 - Tile cuts

Tiles are cut into 7 pieces: a central one and 6 external pieces. This will allow for a tile diversity useful for our final design.

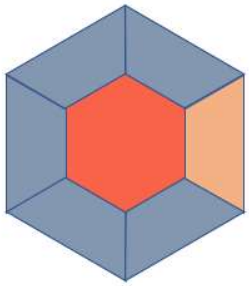
Some of the tiles used in the game:



4 - River start

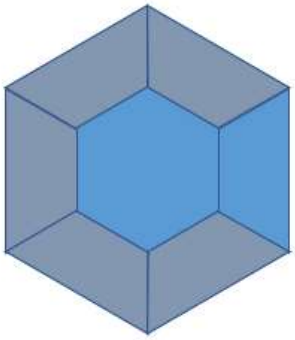


5 - A curving path

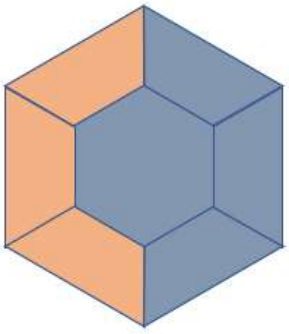


6 - Campfire with a path going there

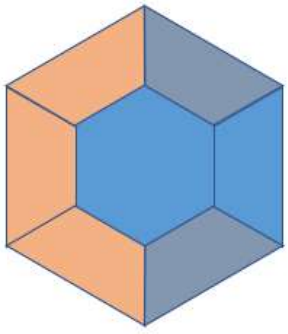
Those tiles are fed to the algorithm. From those other tiles, more complex, can be made.



7 - River

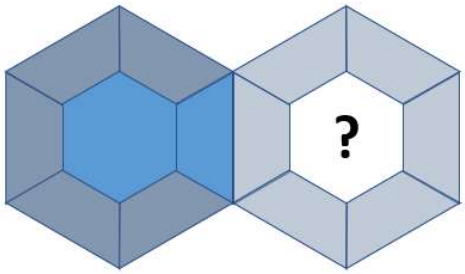


8 - Path



9 - Combined to make a path running along the river

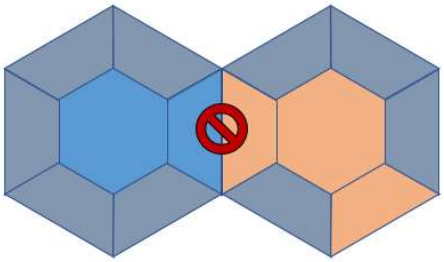
3.2 TERRAIN COMPLETION



10 - Place a new tile

The grid is completed by adding tiles depending on the already placed constraints.

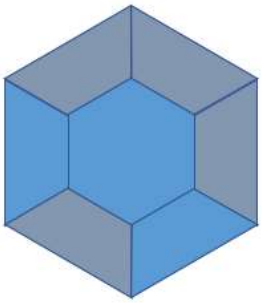
Here for instance, a river tile has already been placed. It now needs to be completed with other tiles.



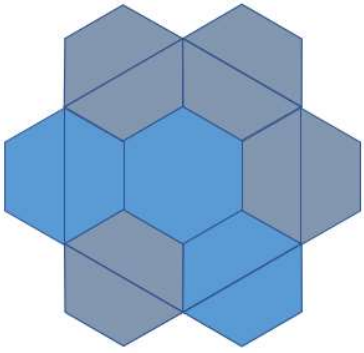
11 - Instance of an illegal placement

A path tile would not be compatible. They do not match.

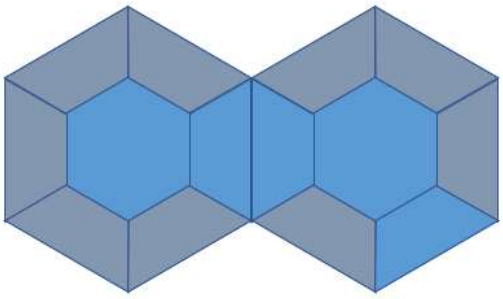
Compatibility is defined by placement constraints. A tile is compatible with a location if the nearby tiles fulfil the adjacence conditions.



12 - A river tile without its constraints

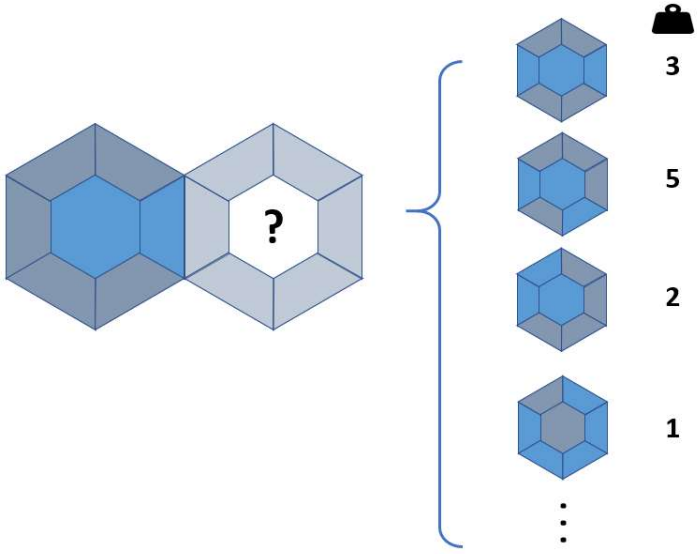


13 - The same tile with its constraints



14 - Instance of a compatible location

Many tiles can be compatible with a same location. The chosen tile is randomly selected using a weight for each tile. Increasing the weight of curved rivers will make them less straight on the final terrain.

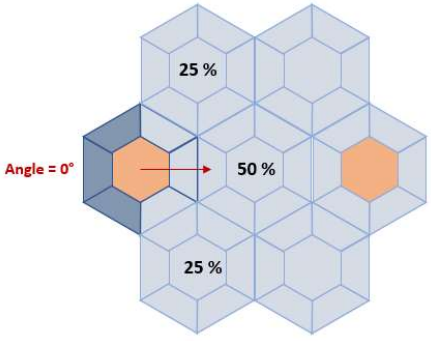


15 - Choosing a tile among the compatible ones

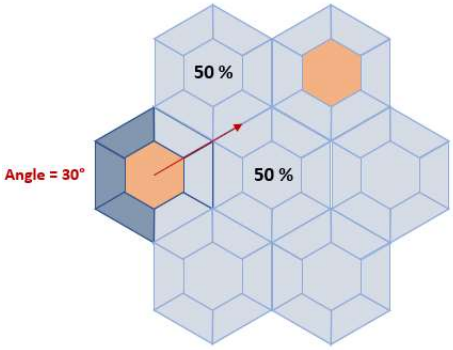
3.3 The Paths

Among the features I want to add, there is the possibility to create paths. They would have a start and an end. The goal is to have a path that would curve a bit and not going straight for the goal.

To make that happen, I start from the starting location and look at the way to go from there.



16 - Choosing the next tile (Case angle=0°)



17 - Choosing the next tile (Case angle=30°)

That direction will be used to figure the next location and the tile to place where I currently am. I keep going until I reach the goal. If there is a dead end, the path stops and raises an exception.

3.4 The rivers

Rivers are a bit special. They grow from a starting location and stop when they reach their maximal length (or if they do not have space to grow further).

4 THE TOOL

4.1 INPUTS

- **The terrain** is a partial grid. Its arrangement must be known by the algorithm (its size and the holes in the grid). The grid format depends on the use case (hexagonal grid, square grid, or… triangular grid? or even non regular grids? *I do not think I will support those, but through generic code, it should be doable*). My API will be using a template for the grid that would need to implement some function (give the neighbours of a location, handle distances, etc.)

- **Tiles** are defined by the user in a `.txt` file. They are encoded with their content, their external constraints, their weight, and a tag allowing the algorithm to identify them when necessary (more on that below).
  Encoding: `[tag] [constraints], [content], [weight]`
  Example: `River R 0 0 0 0 0, R 0 0 0 0 0 R, 5`

4.2 OUTPUTS

- **The modified grid.**

4.3 API

There are 3 main functions:

- ```
  BuildPath(Grid& outGrid,
  Case& start, Case& goal,
  Tag startTag, Tag goalTag, Tag pathTag)
  ```
  That is where the tags come into play. They are used to figure which tile are chosen to start and end a path, and also those in between.

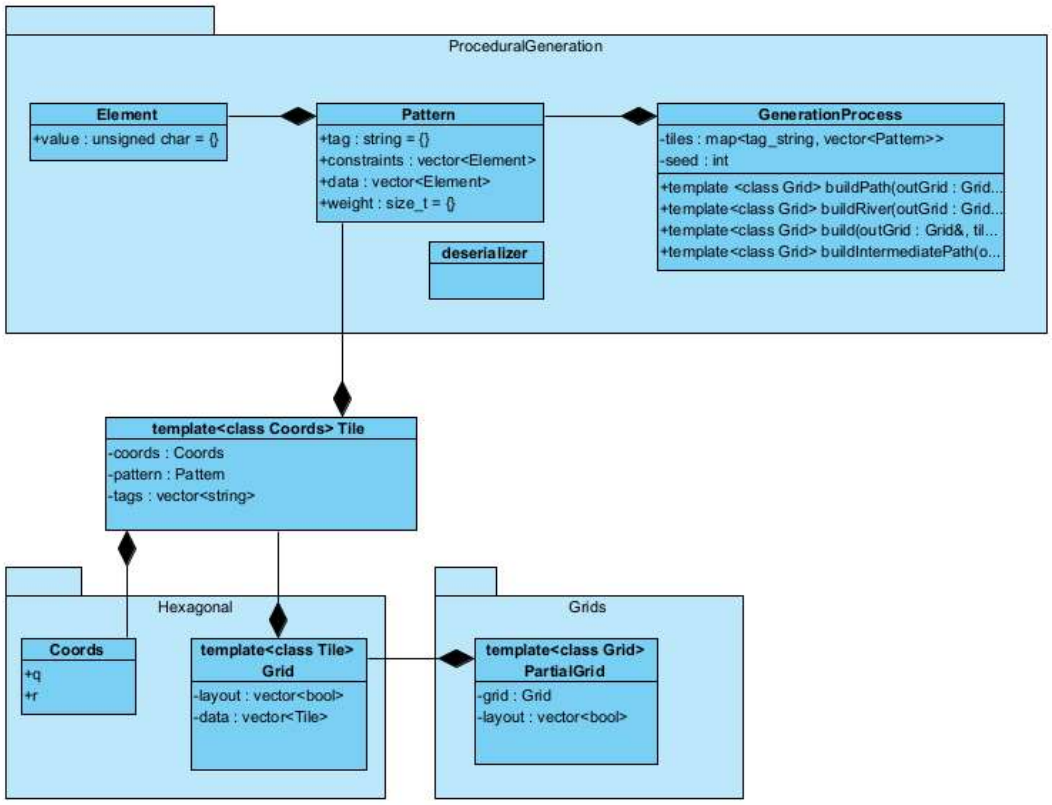
- ```
  BuildRiver(Grid& outGrid,
  Case& start, Size minRiverSize,
  Tag startTag, Tag endTag, Tag riverTag)
  ```

- ```
  Build(Grid& outGrid, Tag tileTag)
  ```
  Use the default algorithm, placing tiles one after the other, starting from the most constrained locations.

Another function is needed for our project:

- ```
  BuildIntermediatePath(Grid& outGrid,
  ConditionType& startCondition,
  ConditionType& goalCondition,
  ConditionType& earlyExitCondition)
  ```
  It allows to create a path whose start and end are fulfilling some conditions. For our specific case, we want to link two paths to one another. An early-exit condition would be to meet another path during the construction. The goal would then be met - we just linked two paths. To make this function more modular, conditions are passed as arguments.

5 CODE (DIAGRAM)



18 - Class diagram for the procedural generation algorithm

The Patterns are created from Elements and deserialized from the text file. They are then used for the procedural generation.

On the other side, the hexagonal grid. It is used by PartialGrid that handles… partial grids!

Finally Tile implements information needed for the procedural generation and is part of the grid.

All functions and methods are not described here. Some would be to select a compatible tile with a location, others to find the next time for a path, etc.

Everything is portable and modules do not depend on one another (except when the procedural generation needs to write the grid).

6 FUTURE IMPROVEMENTS

The input file is a .txt file that is currently completed manually. A possible improvement is to create an external visualisation tool. It would allow the user to see the tiles in the file, add them, remove some, with a friendly visual interface. One could for instance drag and drop tiles content and have a preview of that tile.

Regarding paths, we could use more checks to prevent getting stuck in dead ends and leave the tile in an undesired state.

7 BIBLIOGRAPHY

1. **Dykeman, Isaac.** Procedural Worlds from Simple Tiles. [Online] Octobre 12, 2017. https://ijdykeman.github.io/ml/2017/10/12/wang-tile-procedural-generation.html.

2. **Gumin, Maxim.** Wave Function Collapse. [Online] Octobre 21, 2021. https://github.com/mxgmn/WaveFunctionCollapse.

8 ILLUSTRATIONS