INF5153 - Génie logiciel : conception

TP Génération Procédurale de terrain

Présenté par :	Mokhtar Safir	SAFM14118605
	robitaille-larratt jean p.	ROBJ02049306
	Van Velzen, Philippe	VANP25089705
Groupe :	A	
Remis à :	mosser.sebastien	

Date : 13 décembre 2020

Département d'informatique
Université de Québec a Montréal

Table des figures

Table des matières

1 Introduction

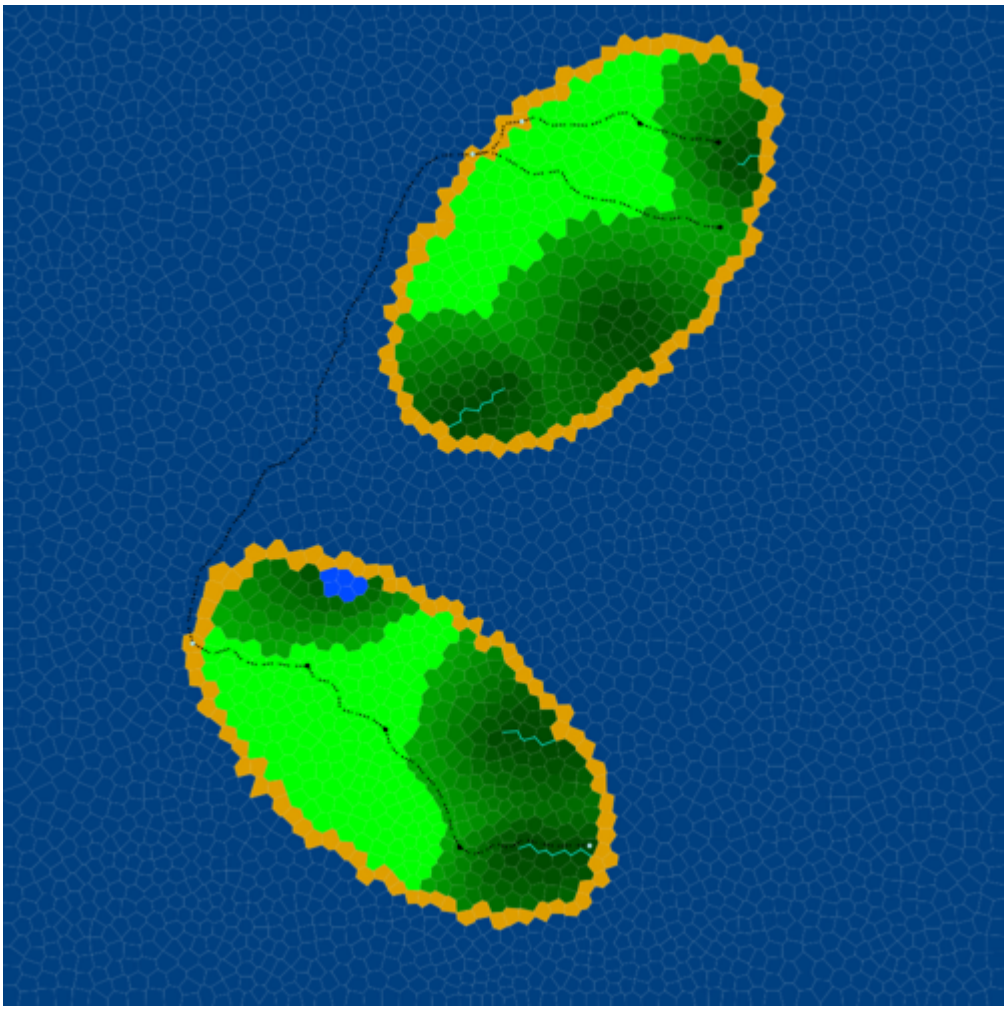
Ceci est le rapport final du projet de session : Génération procédurale de terrain, de l'équipe Swarm-Colony, pour le cours INF5153. Pour résumer le travail que nous avons fait, nous avons présentement toutes les fonctionnalités de la phase 3 : produit final, bien que notre projet ne soit pas sans bug. Ce rapport présente quelques exemples de conception accompagnés de leur diagramme de classe respectif. Nous avons aussi inclue un diagramme de séquence global et une liste de bugs connus et leur explication. Nous terminons par nos bons coups et nos moins bons ainsi que deux exemples d'ajout de fonctionnalité, un ajout facile et un autre plus difficile.

2 Exécution

Commençons par l'exécution. Vous devez exécuter le programme à l'aide des options input et output. Si ces options ne sont pas incluses, le programme lance une erreur. Ensuite, les options suivantes peuvent être utilisées en les combinant de différentes façons et avec les types d'arguments convenable selon les spécifications : shape, water, soil, altitude, heatmap, seed, archipelago, rivers, location, production, pois et roads. L'option pois supporte aussi la possibilité de pouvoir être appelée plusieurs fois. Un exemple d'exécution du programme est la suivante :

./ptg.sh -i samples/sample-1000-1000-4096.mesh -o demo.mesh –shape tortuga –archipelago 2 –pois cities :3 –pois ports :2 –roads –water 1 –rivers 2 –seed 150

Qui devrait générer, après quelques secondes (l'attente est normale), la carte suivante :



Les heatmap (altitude, humidity et ressources) de cette carte sont offertes en Annexe.

Cette carte montre nos îles à leur meilleur, toutes les fonctionnalités y sont, et se génèrent correctement. Ceci n'est pas le cas de toutes les cartes générés. En effet, il arrive par exemple que les rivières n'atteignent pas l'océan. Ceci n'est qu'un exemple des problèmes qui seront discutés plus loin dans ce rapport. Malheureusement, nous n'avons pas développé un système de rejet de carte en cas d'anormalité, mais nous en discutons à la fin du rapport dans la section **Exemple d'ajout de fonctionnalité**.

3 Conception

3.1 Diagramme de séquence global

Nous incluons en Annexe un diagramme de séquence qui montre bien la structure du projet d'un point de vu global. Nous élaborerons sur les différentes classes plus loin dans le rapport, mais ce que nous voulons souligner ici est la structure abstraite qui guidait la conception du projet. Cette structure venait du patron «chaîne de responsabilités».

L'idée était de passer notre objet principal, ici **World**, d'un générateur à l'autre. Chaque générateur modifie le monde selon les fonctionnalités que celui-ci (le générateur) implémente. Cette idée uniformisait l'ajout de fonctionnalité (les fonctionnalité qui modifiaient la carte). Ainsi l'ajout de rivières implique l'ajout d'un générateur de rivière qui s'occupe de modifier notre objet **World** adéquatement.

Ceci permettait aussi la division du travail, chose que nous n'avons pas réellement testé avant la fin du projet, quand nous avons fusionné deux branches contenant 2 fonctionnalités différentes, soit localisation et les points d'intérêts. La fusion a été relativement facile.

Toutefois, cette conception implique que l'objet **World** ressemble beaucoup à une base de données, bien qu'il offre lui aussi des services comme des getter spécifiques (getIsland() par exemple) qui prouvent être très utile lors d'ajout de fonctionnalité. **World** permet aux autres objets de le modifier mais n'exécute pas beaucoup de fonctionnalités. Nous croyons que ce choix est justifié tout de même, puisque qu'il nous a permis d'ajouter de nouvelles fonctionnalités rapidement, facilement et sans conflit avec les autres fonctionnalités.

Nous passons maintenant à quelques exemples de conceptions de plus bas niveau.

3.2 La forme du terrain

Commençons par la forme du terrain. Les terrains peuvent avoir n'importe quelle forme. Pour ce projet, on traite deux formes spécifiques, **un atoll** (forme circulaire) et **une tortuga** (forme elliptique).

Pour cela nous proposons une **interface Shape**. Dans notre cas, la forme **Circle** et **Ellipse** implémente la methode :

```
isInShape(c: Coordinate): boolean
```

qui valide l'appartenance d'une tuile à la forme en se basant sur l'équation mathématique qui définie la forme.

Ensuite, une classe abstraite **IslandShape** qui à son tour est **spécialisé** par une classe définissant la forme finale de l'île. Dans notre cas, la classe **CircularIsland** et **EllipticIsland** mettent en évidence la relation de spécification avec **IslandShape**.

Il est facile de voir que nous pourrions ajouter une île de n'importe quelle forme mathématique.

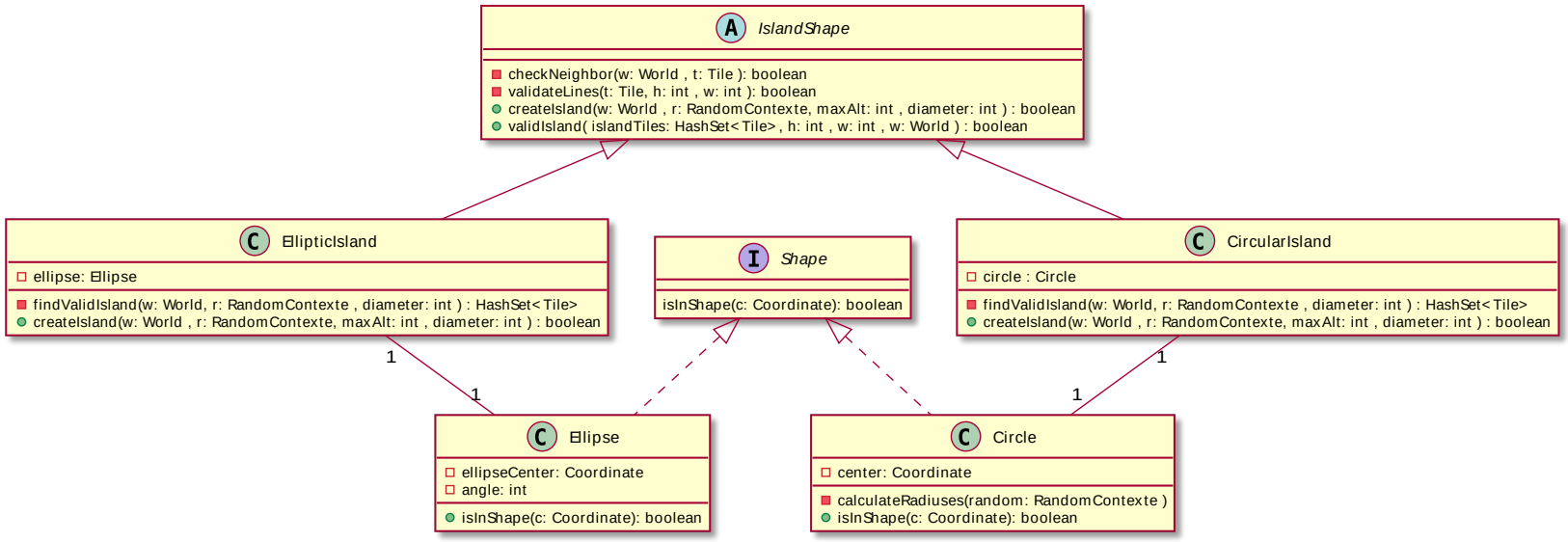


FIGURE 1 – Relation de spécification et de réalisation lors la création d'une forme de terrain

3.3 Le monde

World est notre objet qui contient les informations pertinentes à la créations d'un nouveau mesh. Mais l'objet offre aussi des services pertinent à nos générateurs. Par exemple, le monde contient les informations qui différencient les îles les unes des autres (quand on utilise l'option archipelago par exemple). Ceci facilite l'ajout de caractéristiques, comme des rivières, à chacunes des îles de manière indépendante.

Les tuiles, et les lignes et points qui les composent, sont les unités de base qui permettrons de construire le mesh final. Le monde est un contenant de tuiles et s'occupe d'être le lien entre le mesh (composé de tuiles) et les générateurs (qui modifie les tuiles).

Une autre fonctionnalité démontré dans ce diagramme est la structure du seed, que nous proposons comme étant une stratégie. Cette stratégie étant instanciée seeded ou seedless, si nous avions besoin d'un autre random, pour d'autres fonctionnalités (non-associées à la génération d'île) il suffirait d'en faire un autre. Puisque le random est dans le monde, il peut être appelé dans les générateurs au besoin, puisqu'ils reçoivent le monde en paramètre.

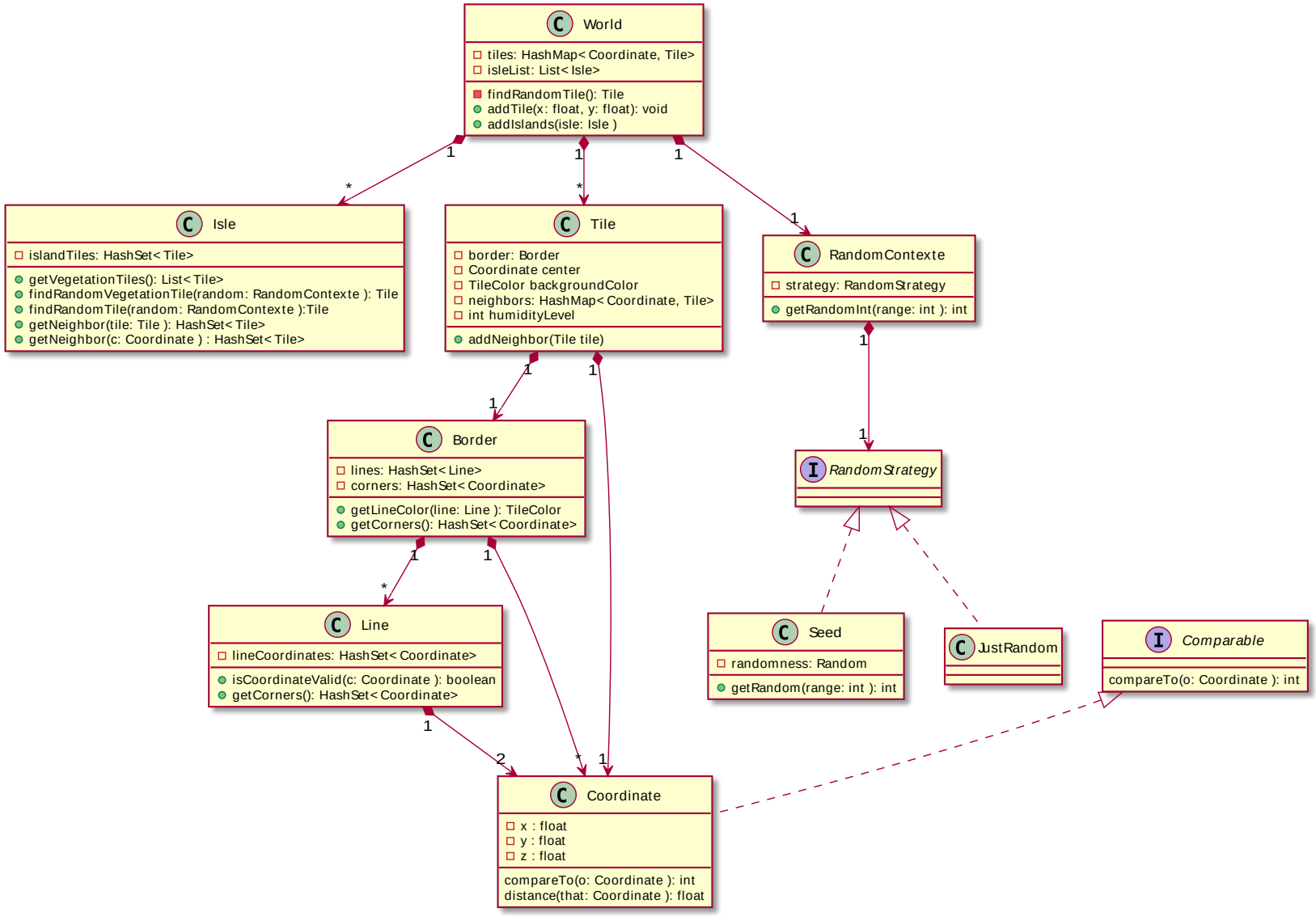


FIGURE 2 – Les différents composant du monde

3.4 Les aquifères

Les aquifères sont des exemples de générateur. Ils sont de type IslandProcessor et ils utilisent la méthode clef de tous les générateurs, la méthode apply(). Ainsi, tous les instances d'aquifère, nape, lake et river, implémente apply. Ils implémentent aussi la méthode spécifique aux acquifères, applyHumidityEffect(), qui elle étalera l'humidité aux tuiles voisines et sera appelée dans apply().

Ce diagramme montre aussi que les lacs et napes implémentent WorldItem, un ajout de la toute fin du projet, qui abstrait la couleur (bleu ou vert foncé) et le type (lake ou Nape). Cette interface intervient aussi dans les biômes, qui sont également représentés par une couleur (jaune) et un type (plage).

Tous les générateurs sont directement ou indirectement appelés dans WorldGénérator. ainsi pour ajouter un générateur, il s'agit d'ajouter une classe qui instancie «Generator» et de l'ajouter dans WorldGénérator. Cette uniformité facilite énormément l'ajout de fonctionnalité.

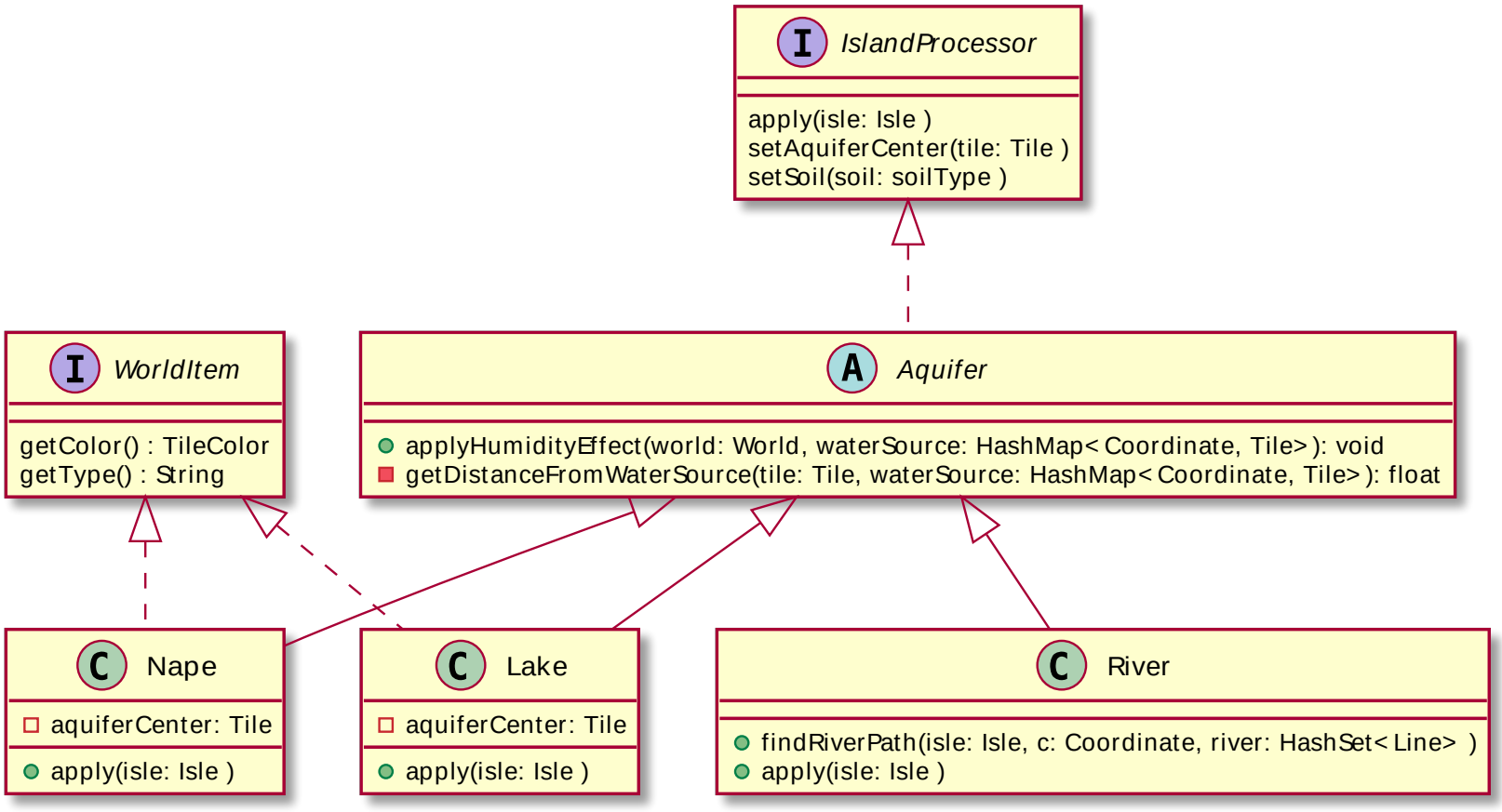


FIGURE 3 – Relation de spécialisation

3.5 Les autres classes

Nous feront un bref survol des autres classes ici.

Package Geometry

Le package de géométrie contient des concepts mathématiques utile à notre projet. Des exemples de classe dans ce package sont Circle, ellipse et coordinate.

Package translator

Ce package contient les méthodes qui éditent le mesh output à l'aide du builder. Il contient aussi la classe qui lit le mesh en input.

Package UserInterface

Ce package contient la classe qui gère les options du programme.

Les autres générateurs

Contenus dans le package World.Generator, ils suivent une structure similaire aux aquifères, mais avec leurs propres particularités. Ces générateurs sont : islandGenerator, biomeGenerator, interestPointsGenerator, roadGenerator et ressourceGenerator.

Autres classes

D'autres exemples de classes, qui sont, selon nous, assez évidentes pour ne pas avoir à être expliquer dans ce rapport sont Tile, Border (la frontière d'une tuile), Isle (une île spécifique).

Finalement nous avons la classe WorldGénérator qui est centrale dans notre projet, elle instancie le monde, elle contient tous les paramètres, elle appelle tous les générateurs et se fait appeler par le meshWriter pour accéder au monde.

4 Points forts et moins fort

4.1 Points forts

Le Mode

Un exemple de conception bien réussi, simple et indépendant du reste, est la conception relié aux Heatmap. Elle n'est pas parfaite, mais l'idée est là. Lors de la synchronisation du mesh (output) avec l'objet **World**, nous passons par le «Mode», qui représente un type de heatmap. Ainsi nous appelons mode.getColor() d'une couleur passée en paramètre et qui est transformé en la couleur du heatmap approprié. Il n'y a pas de répétition de code pour chacun

des Heatmap.

Prenons par exemple la couleur d'une tuile lac dans un Heatmap d'humidité. Le Mode (ici humidity) s'occupe de demander à la tuile ce que ce type de mode veut avoir comme information (l'humidité de la tuile). Elle renvoie ensuite la couleur en lien avec ce Heatmap (bleu) augmentée d'un facteur relié à l'humidité reçue de la tuile.

Les Générateurs

Cette exemple de point fort est déjà décrit plus haut.

4.2 Points faibles

La qualité et l'uniformité du code ont toutefois été légèrement mises de côté dû à un projet compliqué, plusieurs changement de conception et un manque de temps. Ainsi, plusieurs conceptions ne sont pas implémentés uniformément (WorldItem pour les biômes et aquifères). Certaines conceptions ont été laissées à leur première état imparfait puisque ça fonctionnait déjà (Mode pour les Heatmap). Il arrive aussi que nous utilisons des nombres magiques dans notre code.

À cela s'ajoute des bugs qui ne sont pas toujours traités correctement. C'est le thème de la prochaine section. Ajoutons aussi que la gestion des cartes imparfaites (qui devrait être rejetées) n'est pas implémentée. Ceci fera partie d'une section subséquente, où nous essaieront d'implémenter une fonctionnalité facile et une fonctionnalité difficile à notre projet.

5 Bugs connus et potentiels

Nous savons que notre programme comporte déjà quelques bugs. L'exemple le plus évident est l'exemple des rivières qui n'atteignent pas l'océan, elles restent coincées dans des tuiles d'altitudes similaires. Aussi, il arrive que la couleur des lacs est influencée par l'humidité des lacs et rivières avoisinantes.

Nous pensons aussi qu'il y a des bugs potentiels, que nous n'avons jamais rencontrés. Par exemple, nous croyons que si certains arguments sont utilisés de façon absurde (–rivers 1000 disons) cela pourrait être un problème, nous n'avons pas eu le temps de tout tester. Nous garantissons les fonctionnalités avec des nombres acceptables. Souvent des îles trop petites où il n'y a que de la plage empêche l'ajout d'autres caractéristiques aux îles, ce qui est également un problème.

6 Exemple d'ajout de fonctionnalité

Nous concluons ce rapport en vous proposant brièvement deux ajouts de fonctionnalité que nous trouvons intéressant et qui démontrent un peu, espérons-le, la justesse de notre conception.

6.1 Fonctionnalité facile

Nous croyons qu'il serait relativement facile d'ajouter un log et un chronomètre pour chacunes des étapes de la génération des îles, similaire aux «load screen» de jeux. Ceci est un exemple de fonctionnalité que M. Mosser proposait d'ajouter de façon théorique lors de la présentation orale de notre projet.

Puisque notre projet est divisé en générateur, il serait facile de logguer chacunes des étapes dans un fichier ou au terminal. Cependant, tout dépendant de la précision des étapes qu'on désire logguer, on aura soit à ajouter ces lignes de code dans notre WorldProcessor (ce qui serait facile) ou dans chacun de nos petits générateurs (plus de code et donc plus compliqué à gérer).

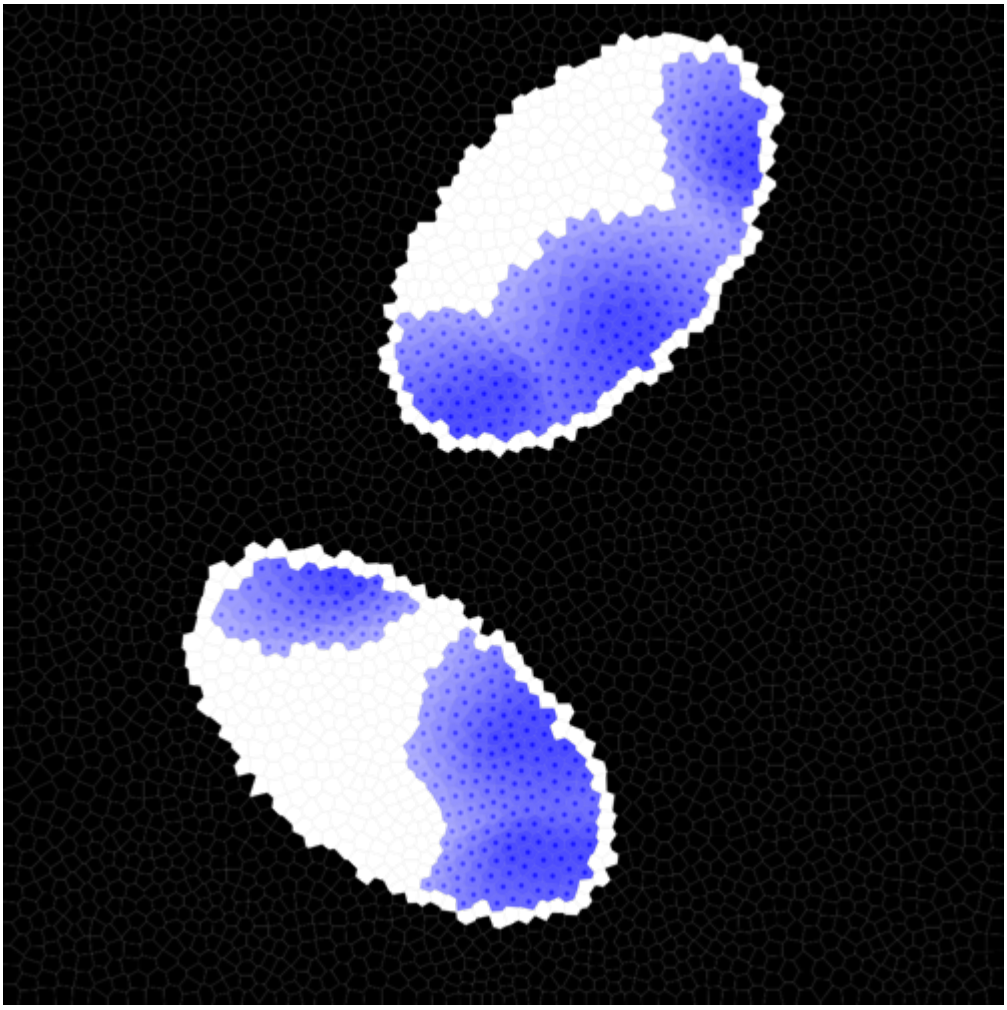
Nous croyons donc que notre projet permet l'ajout de cette fonctionnalité.

6.2 Fonctionnalité plus difficile

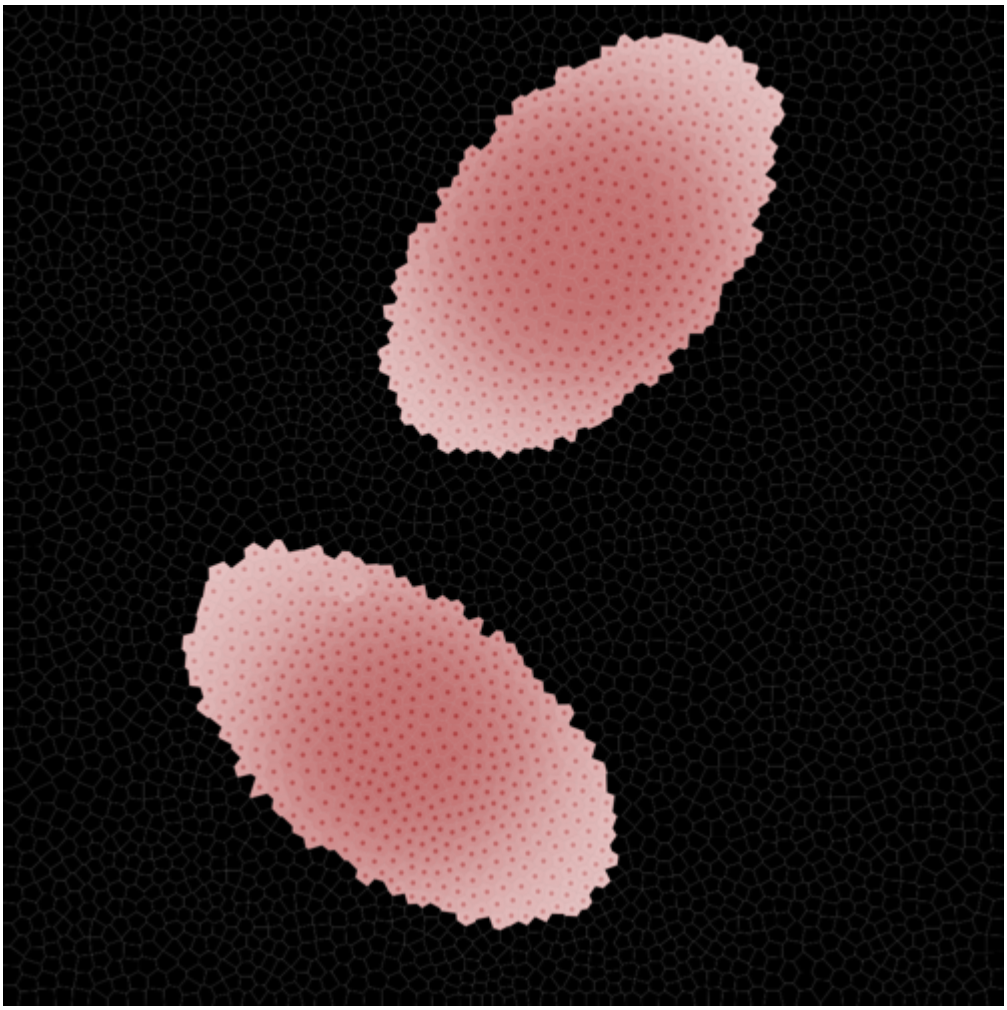
Ajouter une fonctionnalité qui trie les bonnes cartes des mauvaises serait une fonctionnalité que nous pourrions ajouter mais non sans difficultés. En effet, il faudrait déterminer une façon de trier les cartes. Cependant, nous avions eu l'idée d'ajouter à notre interface de générateur, en plus de la fonction apply(), la méthode verify() qui vérifierait, de façon plus globale, l'aspect réaliste de notre carte. Ainsi, après chaque apply(), on exécuterait verify() et on rejetterait les cartes non réalistes.

Nous pouvons donc facilement placer les vérifications, mais l'algorithme de vérification lui même serait possiblement difficile à implémenter. Nous croyons tout de même que ce serait possible.

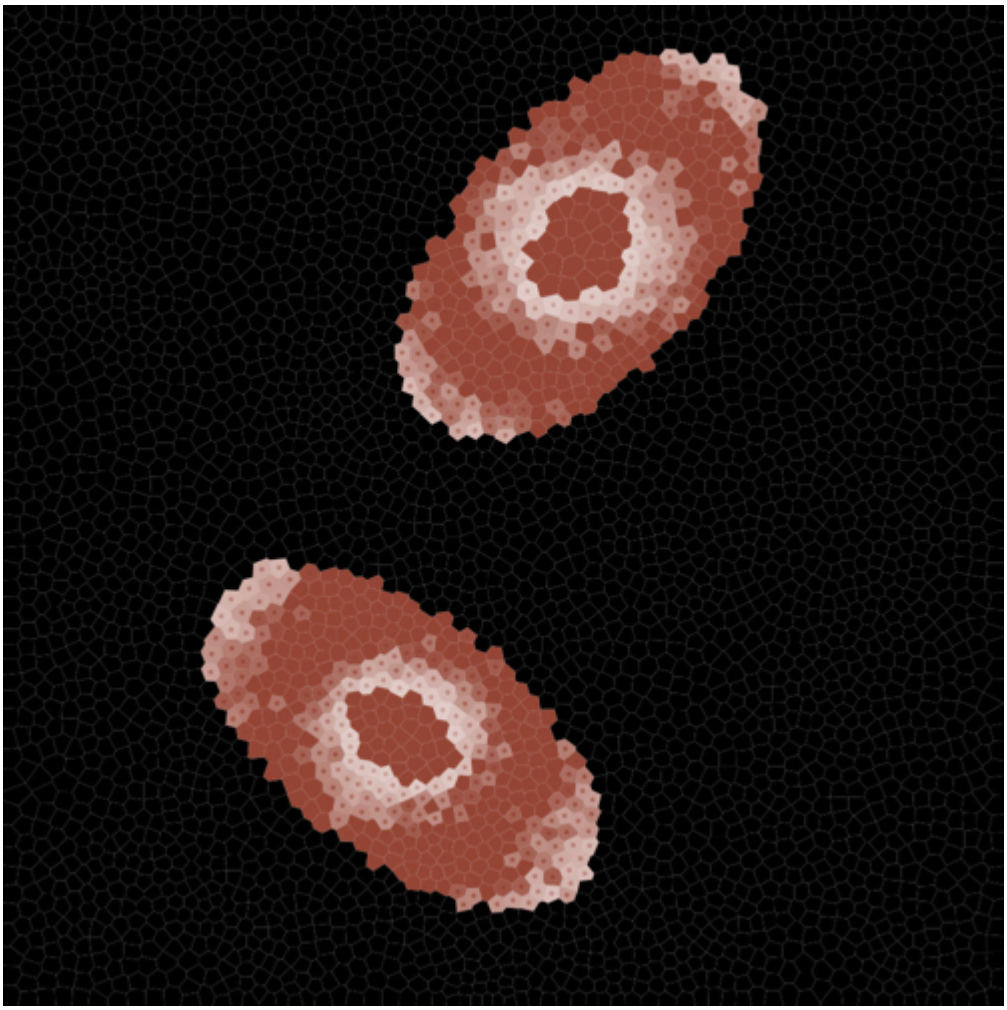
7 Annexe



Heatmap d'humidité de la carte présentée au début.



Heatmap d'altitude de la carte présentée au début.



Heatmap des ressources de la carte présentée au début.

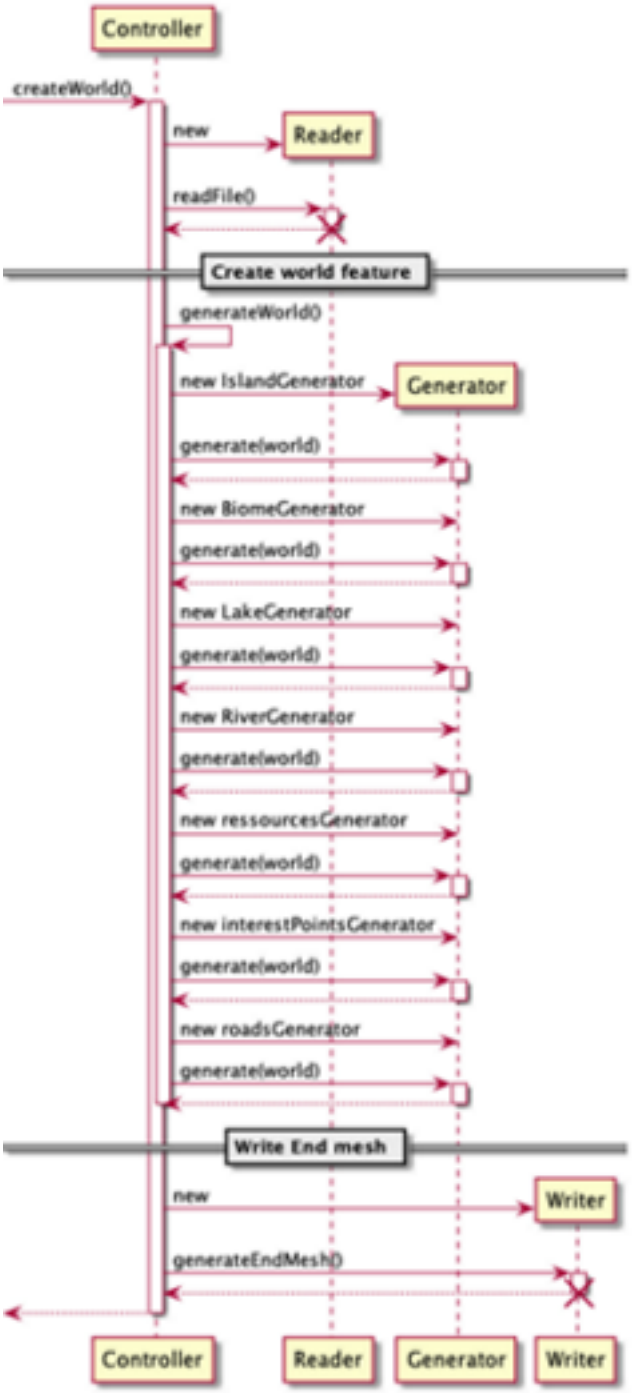


Diagramme de séquence global du projet.