Rapport - Projet de session : Génération procédurale de terrain

INF5153 - A20 - UQAM

Travail réalisé par: Frederic Alas (ALAF02039606) et Nahuel Londono (LONN17079800)

Présentation

L'implémentation d'un générateur procédural de terrain présentée dans le cadre du projet de session du cours INF5153 - Génie Logiciel: conception s'inscrit dans un cadre académique. Celle-ci se base sur un générateur de maillage fourni et ajoute un nombre de fonctionnalités de création de terrain comportant différents attributs tels que l'ajout de plans d'eau, la présence de biomes ou bien l'incorporation de la notion d'altitude. L'implémentation devait utiliser les concepts de programmation orientée objet vu dans le cadre du cours. Dans les paragraphes suivants, les choix de conceptions seront présentés et justifier de façon à faciliter la compréhension du projet pour toute personne s'intéressant à celui-ci.

Avant la lecture, il est important de prendre en compte que les différents diagrammes pertinents à ce projet se retrouvent à la fin de ce document. Pour une meilleure visualisation de ces derniers, on vous recommande d'aller voir les fichiers .svg pouvant être trouvés dans le folder diagrammes du projet.

Phase 1 - Modèles de conception

La phase 1 reposait principalement sur l'adaptation du générateur de maillage fourni en un code orienté objet, qui servirait de point de départ pour le reste du projet. Les fonctionnalités ajoutées lors de cette phase étaient celles liées à la forme de l'île générée (atoll et tortuga). Pour se faire, l'implémentation proposée se base sur le patron de conception `proxy` afin d'enrober certaines notions du générateur fourni. Tout d'abord la notion de polygon a été enrobée dans la classe Tuile afin d'éliminer la dépendance directe à la notion de `Polgyon`, qui pourrait être modifiée sans nécessairement changer les interactions des Tuiles avec le reste de l'application. Par la suite, un ensemble de Tuiles composent un Terrain afin de représenter un maillage qui comporte un nombre d'attributs nécessaires à la bonne construction de celui-ci. L'utilisation du patron de conception `factory` permet d'éliminer la complexité de la création du maillage en passant par la composition d'un Terrain d'un ensemble d'objets peu complexes responsables des spécifications d'instanciation. Afin de représenter la notion de forme de terrain, l'utilisation d'une classe Shape a été utilisée, celle-ci supporte (pour la phase 1), la forme `atoll` seulement qui peut être appelée de façon à permettre la variation de la forme selon le paramètre reçu. En plus de la fonctionnalité de forme, la phase 1 apporte la notion de biomes, le placement et la coloration de ceux-ci. L'utilisation de la fabrique semblait pertinente puisque chacune des cases comporte un biome (d'où l'utilisation d'une classe abstraite Biome, étendue par des types plus précis), donc l'utilisation d'une même mécanique d'instanciation a été implémentée en adaptant le type de biome appliqué sur la Tuile.

L'idée principale derrière l'implémentation initiale est la fragmentation des responsabilités des concepts nécessaires à la construction d'un terrain simple (seulement les paramètres de base) et de placer la structure nécessaire à l'ajout de nouvelles fonctionnalités.

Phase 2 - Produit minimal viable

La phase 2 avait pour principal but l'ajout de fonctionnalités basique permettant la variation du terrain généré et du raffinement de certaines implémentées à la phase 1. L'implémentation proposée génère un terrain de base, sans aucune variation provenant des options, pour ensuite appeler un nombre de fonctions ajoute celle-ci au besoin. Les fonctionnalités ajoutées sont appelées à l'aide d'option lors de la génération de terrain. L'idée derrière ce mode de fonctionnement est de limiter les interdépendances entre les options de façon à rendre celles-ci modulaire. La séparation faite vise à permettre de multiples combinaisons d'options sans que celles-ci n'aient un impact la création basique d'un terrain. De plus, la seconde phase permet la création de terrain des formes suivantes: atoll (un beignet) et tortuga (île sur le long), chacune de ces formes nécessite un traitement spécifique afin de placer correctement les biomes (couleurs) parmi les cases.

La fonction d'altitude, appelée par l'option `-altitude`, permet de contrôler la hauteur maximale du terrain au moyen d'un entier numérique. Comme mentionné plus haut, la forme de l'île peut avoir un impact sur le traitement des cases. La fonction gérant l'altitude prend donc la forme en paramètre et ajuste l'application de la hauteur en fonction de celle-ci sans nécessiter de validation préalable.

Ensuite, les options `-water` et `-soil` permettent l'ajout d'un nombre de plans d'eau spécifié par l'utilisateur et placer aléatoirement sur la carte. Au même titre que l'option précédente, celles-ci peuvent être appelées indépendamment des autres options. Ces options introduisent la notion d'humidité des biomes, notions qui n'a aucun impact sur l'implémentation précédente tout en ajoutant des informations visibles sur le terrain généré.

La troisième fonctionnalité ajoutée durant la seconde phase est celle de `-seed` qui permet à l'usager de contrôler sortie du générateur. Cette option utilise un entier numérique comme base à toute option aléatoire afin d'avoir un terrain identique à chaque itération. Cette option peut donc avoir un impact sur celles spécifiées plus haut d'un point de vue client, sans avoir d'impact au niveau de l'implémentation puisque les principes de base de la génération de terrain restent les mêmes. L'ajout des fonctionnalités `-soil` et `-altitude` étant conflictuelle lors de la coloration du terrain.

L'ajout de l'option `-heatmap` permet de prioriser l'une ou l'autre de ces options. La génération incrémentale du terrain permet de brancher sur une ou l'autre des colorations possibles en fonction de l'argument de l'option `-heatmap` sans avoir d'impact sur l'implémentation existante.

L'utilisation du principe ouvert/fermé permet nous a permis d'étendre les fonctionnalités des classes existantes sans modifier les comportements précédents facilement. De plus, selon le même principe, il nous a été possible d'utiliser les fonctionnalités de cette phase pour complexifier davantage l'implémentation du générateur de terrain lors de phase future.

Vue générale du projet

L'implémentation, dans sa forme actuelle, est loin d'être optimale d'un point de vue orienté objet. Une grande portion des responsabilités sont attribuées aux classes `Tuile` et `Terrain`, ce qui introduit un couplage fort au sein de l'implémentation. Ce couplage fort a, dans notre cas, entrainé un effet de "code spaghetti" et limiter l'indépendance de chacune des fonctionnalités offertes.

L'absence de répartition des responsabilités a rendu plus complexe le développement puisque la plupart des fonctionnalités ajoutées venaient briser celles existantes lors du merge des branches de développement avec la version stable de `Master`. Évidemment, ces difficultés vont autant être présentes, voire pires, lors de la maintenance future d'un logiciel construit comme celui-ci. Le problème présenté dans les lignes précédentes s'est principalement manifesté lors de l'implémentation des fonctionnalités de la phase 2, ce qui peut indiquer que certains choix de conceptions faits lors de la première phase auraient dû être revu au lieu de tenter de contourner ceux-ci comme ça a été le cas. L'utilisation des classes "fourre-tout" s'est intensifiée tout au long du développement.

Dans le contexte actuel, il était impossible de déconstruire ce qui était fait afin d'offrir une implémentation plus alignée sur les principes de conception orientée objet. Une implémentation plus modulaire aurait pu permettre une évolution des fonctionnalités plus indépendantes les unes des autres. Une indépendance pour chacune des fonctionnalités permet une séparation "naturelle" des composantes du logiciel en différent sous-dossiers (packages) regroupant le code nécessaire à un aspect du logiciel, facilitant ainsi les développeurs non familiers avec l'application à se retrouver dans le code et donc, facilite la maintenance.

De plus, l'implémentation actuelle est fortement dépendante de la forme du maillage utilisé, malgré une tentative d'implémenter du patron `proxy` comme mentionné dans les paragraphes ci haut; cette dépendance expose le logiciel à un bris complet si le type de maillage venait à changer. Des aspects spécifiques au maillage sont utilisés à plusieurs reprises dans les classes `Terrain` et `Tuile` suggère un mauvais découpage des celles-ci en plus d'exiger une compréhension du maillage sans quoi il sera plus difficile d'ajouter au logiciel.

L'importante interdépendance des composantes de l'implémentation complexifie grandement la possibilité de tester chacune d'entre elle séparément. C'est d'ailleurs pourquoi le nombre de tests automatisés est si bas et que les tests n'assurent pas l'intégrité du logiciel. Idéalement, une conception complètement orientée objet permet d'implémenter une suite de tests unitaires couvrant une importante partie des fonctionnalités de celui-ci et assure qu'un changement n'impacte pas négativement les composantes préexistantes (dans un cas où les tests sont suffisants et biens construits).

Si le projet était à refaire, il aurait été souhaitable de découper l'application en plus petit sous modules plus fidèles aux patrons de conception GRASP vu dans le cadre du cours. Par exemple, l'implémentation plus présente et mieux définie d'un polymorphisme permettant à au terrain généré d'avoir une forme spécifiée par l'utilisateur sans devoir passer par une multitude de validations interdépendante comme l'implémentation actuelle le fait. Une classe de type `expert de l'information` aurait pu permettre d'assigner la responsabilité d'une action à la classe adéquate et ainsi, limiter la faible cohésion de la classe `Terrain`. De plus, l'application plus présente des principes `SOLID` aurait permis un développement itératif plus fidèle aux bases d'une application orientée objet. Comme mentionné à plusieurs reprises, l'interdépendance des composantes du logiciel vient directement à l'encontre du principe d'inversion des dépendances puisqu'une grande partie du fonctionnement du logiciel est basée sur un ou plusieurs détails au lieu d'être basé sur des abstractions. Suivant le même ordre d'idée, plusieurs classes de l'implémentation proposée se base sur de grande classes (gérant plusieurs responsabilités) et très peu sur de petites interfaces tel que recommande le principe de ségrégation des interfaces. Ce défaut se fait davantage remarquer lorsque vient le moment d'ajouter de nouvelles fonctionnalités à l'application et qu'un développeur se butte à une immense classe ayant un rôle peu clair.

Malgré les difficultés rencontrées, le développement d'un projet tel que celui-ci permet de comprendre l'importance d'une conception solide dans un contexte où l'application se base sur un ou des concepts préexistants.

Diagramme de classes

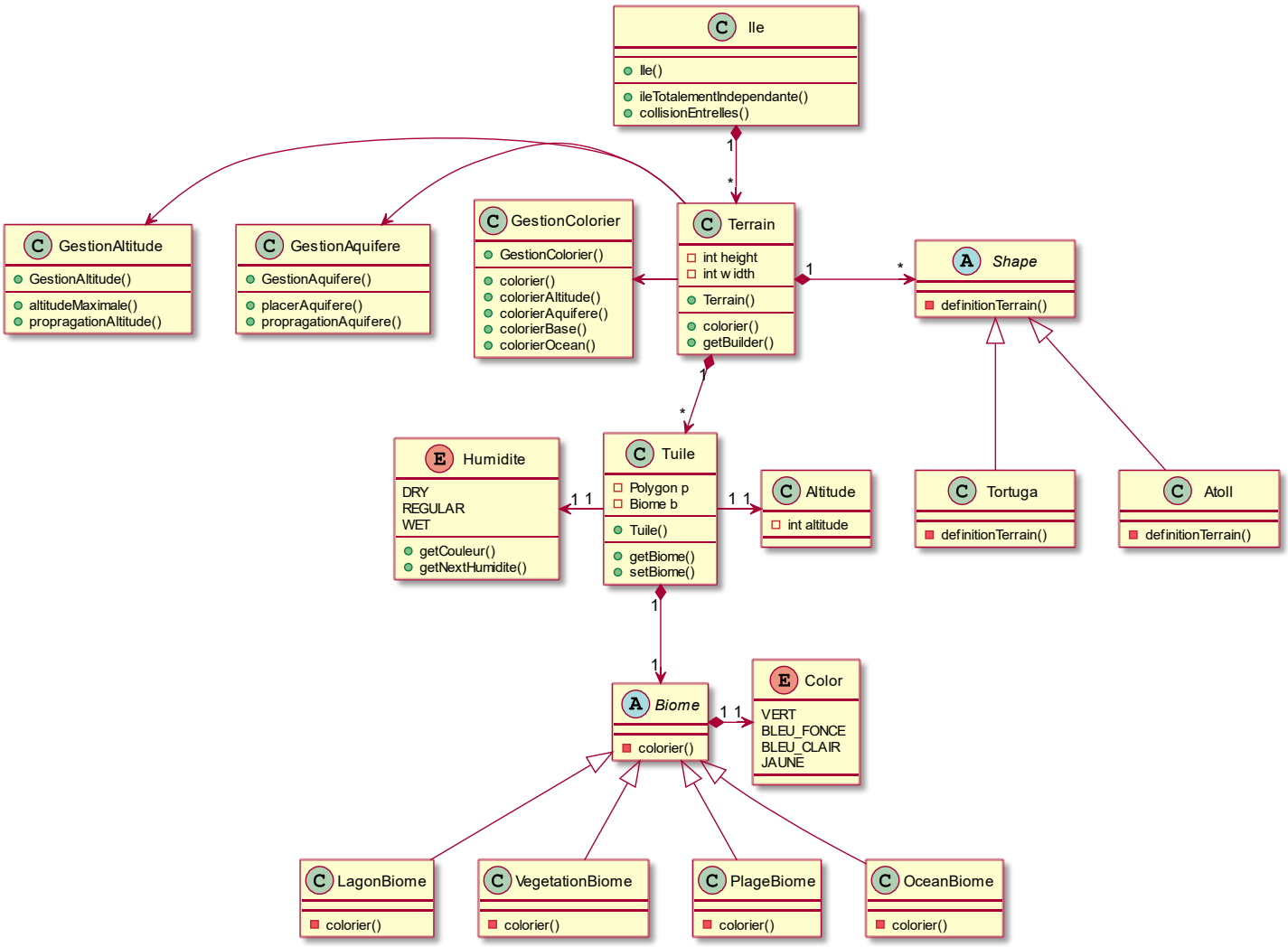

Diagramme de séquence (vue spécifique pour définition biomes atoll)

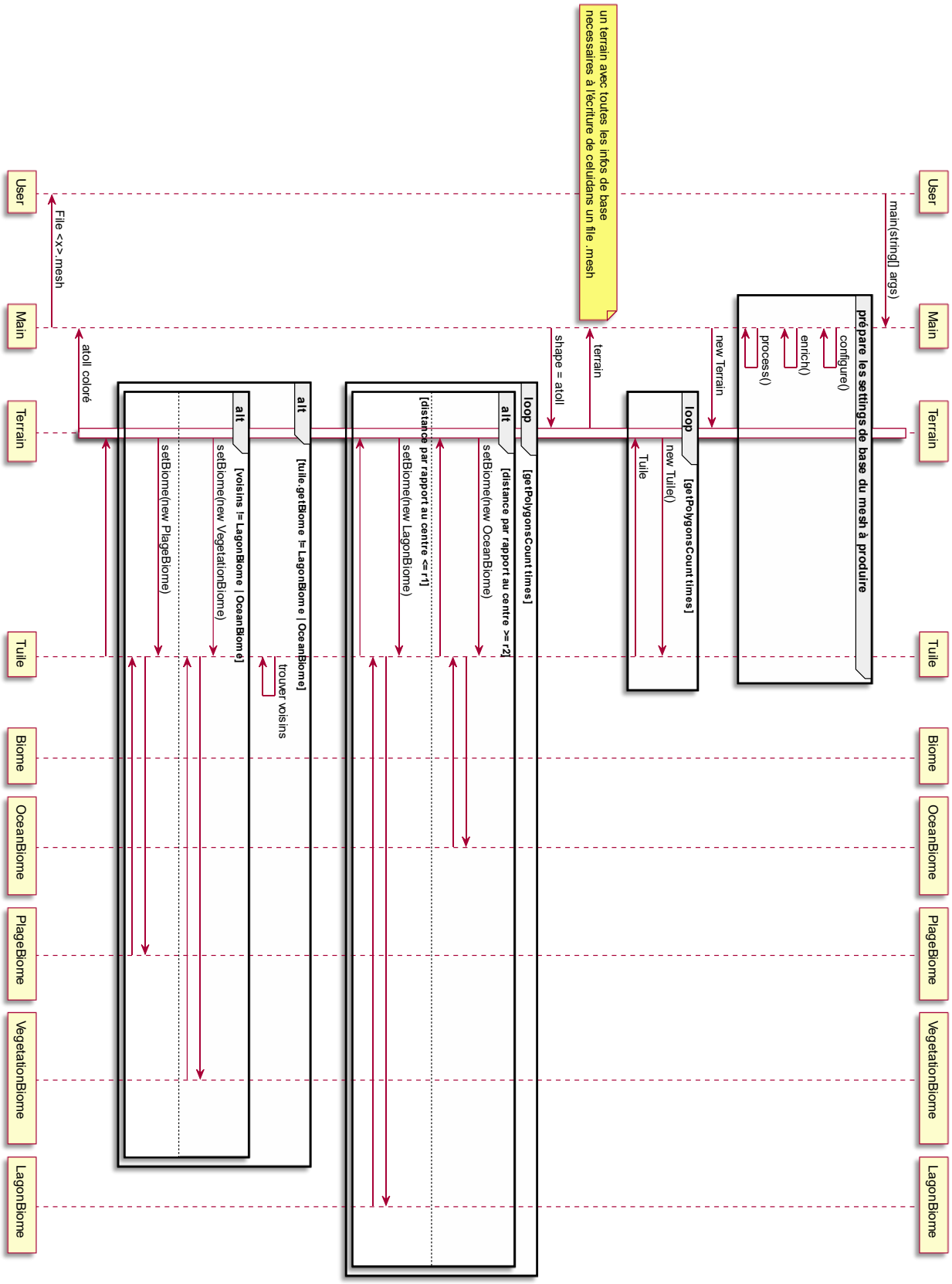

Diagramme de séquence (vue globale)