Université de Bordeaux

Département Informatique

Année 2019-2020

Groupe 21-2

Master 1

Projet de Programmation: Génération procédurale de planètes sphériques

Mémoire

Client:
David Renault

Auteurs:
Rémi Barbosa
Benjamin Darmet
Marc Cerutti
Sofian Antri
Tsiory Rakotoarisoa

Sommaire

1 Introduction

En informatique, la génération dite procédurale est la création de contenu numérique (modèle 3D, animation, son, musique,...) à une très grande échelle et de manière automatisée. Elle doit répondre à un ensemble de règles qui sont définies par des algorithmes.

Nous cherchons à mettre en place au sein de ce projet, un outil permettant de générer procéduralement et de visualiser des planètes (une à la fois) dans un but ludique (se balader pour le plaisir d'admirer et explorer, découvrir un nouvel environnement). Chaque planète est représentée par une sphère géodésique (divisée en faces) statique dont les faces colorées témoignent de la présence de différents terrains, tels que des océans, des reliefs, des déserts, des plaines etc. Un exemple est visible sur la figure 1. La planète aura un rendu "low-poly", désignant un style graphique avec peu de polygones et donc volontairement peu de détails pour des raisons d'optimisation mais aussi d'esthétique. Une fois la planète générée, il sera possible de la visualiser par une vue à 360° en orbite autour de la planète.

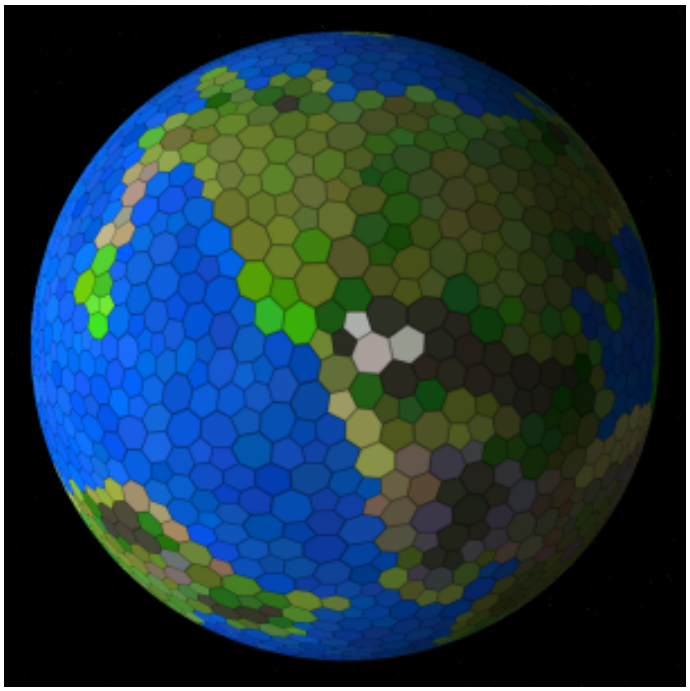

Figure 1: Exemple de planète générée procéduralement[1]

[1]https://experilous.com, Procedural Planet Generation Posted on September 30, 2014 by Andy Gainey. Last access : 05/04/2020.

2 Analyse de l'existant

2.1 Algorithmes de génération de sphère

2.1.1 Subdivision d'un icosaèdre : icosphére

Il existe différentes méthodes pour générer une sphère à face polygonale. La toute première est la subdivision à plusieurs reprises d'un icosaèdre pour obtenir une géode. Un icosaèdre est un polyèdre composé de 12 sommets et 20 triangles équilatéraux (visible figure 2). La géode quant à elle est un polyèdre convexe inscrit dans une sphère dont il réalise une approximation. Pour obtenir cette géode, on réalise une subdivision des triangles composant l'icosaèdre en 4 triangles plus petits. On répète cette opération plusieurs fois afin d'obtenir la surface désirée. On peut voir le résultat après 2 subdivisions consécutives sur la figure 3.

L'utilisation de cette approche permet d'obtenir une position des points régulière mais aussi une décomposition de la sphère en triangle de même taille sans appliquer d'autres algorithmes supplémentaires (Comme Voronoï ou Delaunay qui sont expliqués plus bas). Il s'agit de la méthode standard. Il est possible d'appliquer cette subdivision sur les uv-spheres et les cubesphéres, mais les polygones ne seront pas identiques, et donc leur taille non plus. Pour plus d'informations, voir la référence [1]. Il est néanmoins possible de partir directement de points et de les relier par la suite. C'est le cas de l'utilisation de la sphère de Fibonacci.

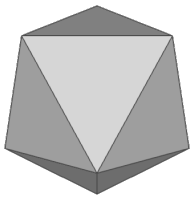

Figure 2: Icosaèdre[2]

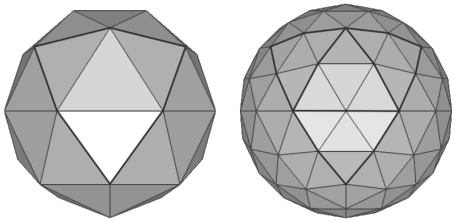

Figure 3: Subvidision à 2 reprises d'un icosaèdre.[3]

2.1.2 Sphère de Fibonacci

Une sphère de Fibonacci est la représentation d'une sphère par dispersion de points (voir figure 4). Différents algorithmes peuvent être utilisés pour générer cette répartition de points de manière plus ou moins égale. Il y a par exemple celui de la répulsion electrostatique. Il consiste à partir d'une distribution arbitraire des points à appliquer un algorithme de répulsion ponctuelle, dans lequel tous les points se repoussent les uns des autres. On fait alors tourner l'algorithme jusqu'à un certain critère de convergence, autrement dit quand la distribution nous convient. Pour plus d'informations voir [5].

[2]http://www.songho.ca/, OpenGL Sphere, 2018 by Song Ho Ahn, Accessed January 2020

[3]http://www.songho.ca/, OpenGL Sphere, 2018 by Song Ho Ahn, Accessed January 2020

[4]https://www.redblobgames.com, Delaunay + Voronoi on a sphere Posted on March 10, 2019 by RedBlobGames, Accessed January 2020

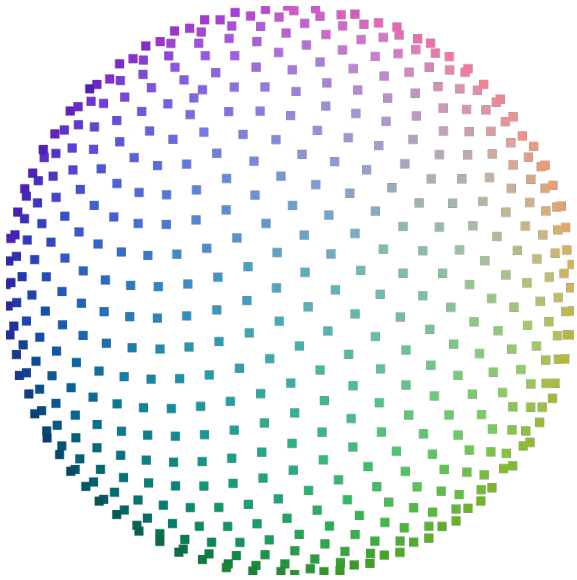

Figure 4: Sphère de Fibonacci [4]

Une fois la répartition des points obtenue, il est nécessaire de les relier afin d'obtenir une première version d'une sphère. On peut alors utiliser le tout premier algorithme : La triangulation de Delaunay. Cette triangulation consiste à relier tous les points d'un plan P, tel qu'aucun point de P n'est à l'intérieur du cercle circonscrit d'un des triangles formés. On peut bien sûr étendre cette définition à un espace en 3 dimensions. On parle alors de sphères circonscrites. La sphère finale obtenue est celle de la figure 5.

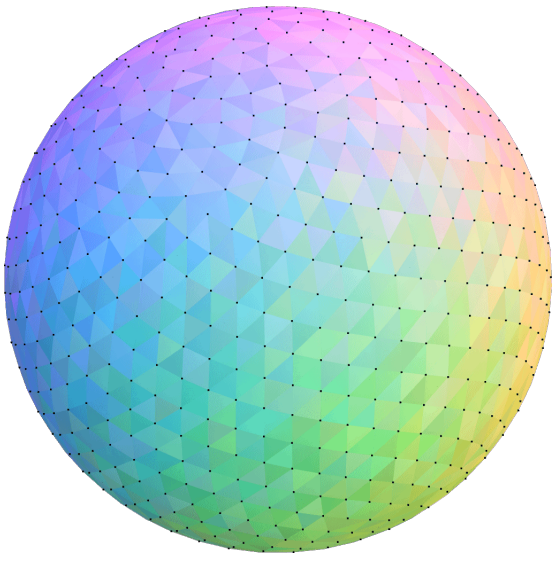

Figure 5: Résultat de la triangulation de Delauney pour une sphère[5]

Il est alors facile d'appliquer le diagramme de Voronoï par la suite. Ce diagramme consiste à un pavage (découpage) du plan en cellules. Chaque cellule contient un "germe", représenté par un point, dont la cellule en représente sa zone d'influence. Les sommets du diagramme de Voronoï sont les centres des cercles circonscrits des triangles obtenus par la triangulation de Delaunay. Les arêtes du diagrammes quant à elles sont les médiatrices des arêtes de la triangulation de Delaynay.

[5]https://www.redblobgames.com, Delaunay + Voronoi on a sphere Posted on March 10, 2019 by RedBlobGames Last access : 05/04/2020.

On peut voir la superposition d'un diagramme de Voronoï et de la triangulation de Delaunay à la figure 6. L'application sur une sphère est présentée à la figure 7. Pour plus de détails, il est possible de voir [6].

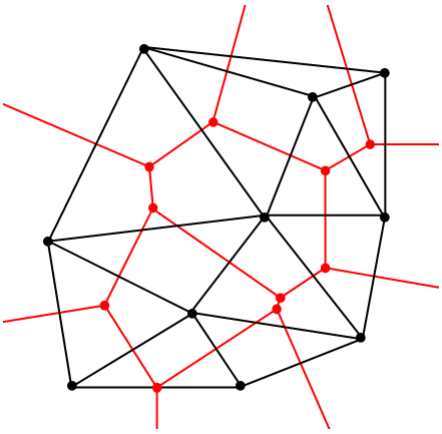

Figure 6: Superposition d'un diagramme de Voronoï (en rouge) et de la triangulation de Delaunay (en noir).[6]

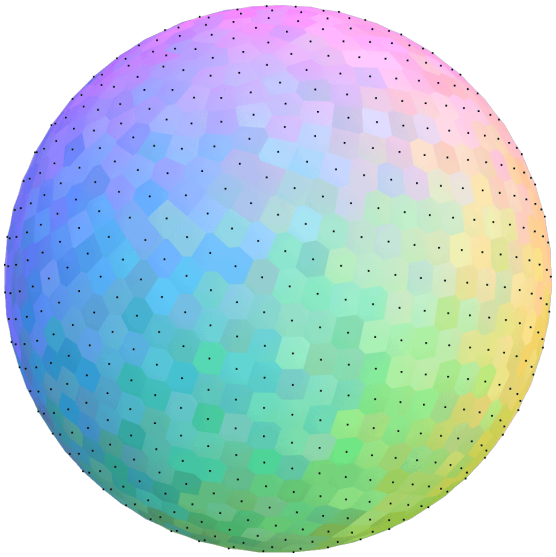

Figure 7: Résultat de l'application d'un diagramme de Voronoï sur une sphère.[7]

[6]https://www.redblobgames.com, Delaunay + Voronoi on a sphere Posted on March 10, 2019 by RedBlobGames Last access : 05/04/2020.

[7]https://www.redblobgames.com, Delaunay + Voronoi on a sphere Posted on March 10, 2019 by RedBlobGames Last access : 05/04/2020.

2.2 Algorithmes de générations procédurales aléatoires

Il existe des algorithmes souvent utilisés dans l'industrie du jeu vidéo et de la modélisation 3D permettant de générer procéduralement à partir de textures avec une part d'aléatoire, des environnements plus ou moins réalistes comme des terrains, des nuanceurs ("shader" en anglais), etc. Un exemple est présent sur la figure 8.

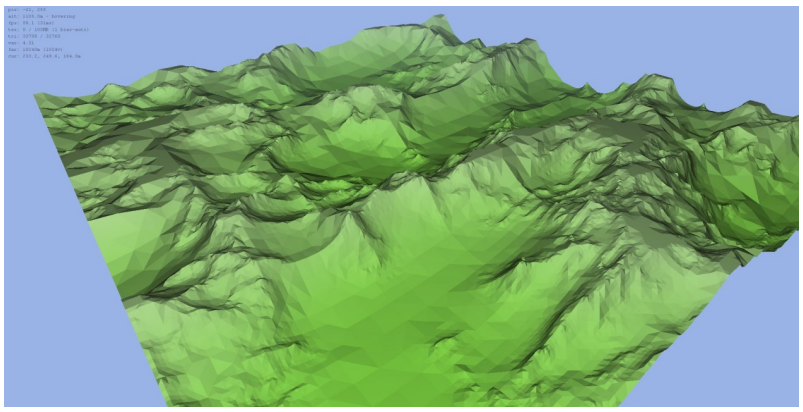

Figure 8: Image de terrain généré procéduralement[8].

Ces algorithmes se basent sur le modèle du "bruit" mathématique utilisé dans la théorie des signaux, mais avec des paramètres qui permettent de varier les effets. C'est ce qu'on appelle le "chaos", de l'aléatoire contrôlé.
Un exemple très simple de bruit serait celui dit "blanc"(figure 9), ne faisant pas partie de ces bruits chaotiques, il est purement aléatoire et se retrouve dans de nombreux domaines. Il est peu intéressant pour générer des environnements réalistes, et c'est pour cela que l'on écartera tous les bruits que l'on ne peut pas contrôler.

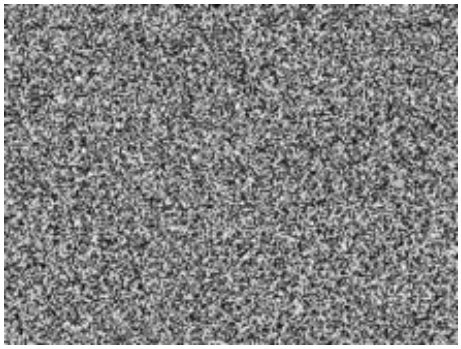

Figure 9: Image de bruit blanc[9].

Il reste trois types de bruits qui peuvent être intéressant pour ce projet : les bruits de valeurs, les gradients de bruits (Perlin Noise, Simplex Noise), et les bruits organiques (Voronoï, Worley) que l'on peut voir figure 10. Cependant vu la diversité des différents bruits qui existe, le plus intéressant est le bruit de Perlin. C'est d'ailleurs le plus utilisé pour réaliser des reliefs de terrain. Il a l'avantage d'être très flexible sur les paramètres donnés, tout en produisant des textures réalistes. Il existe aussi une amélioration moins coûteuse en ressources, Simplex Noise, et qui fait moins d'anormalité sur le rendu 3D (artéfact). Pour plus d'informations, lire [11].

[8]https://sainarayan.me, Procedural Terrain Generation Posted on May 13, 2015 by Cyber Shaman
Last access : 05/04/2020.
[9]Image from wikipedia
Last access : 05/04/2020.

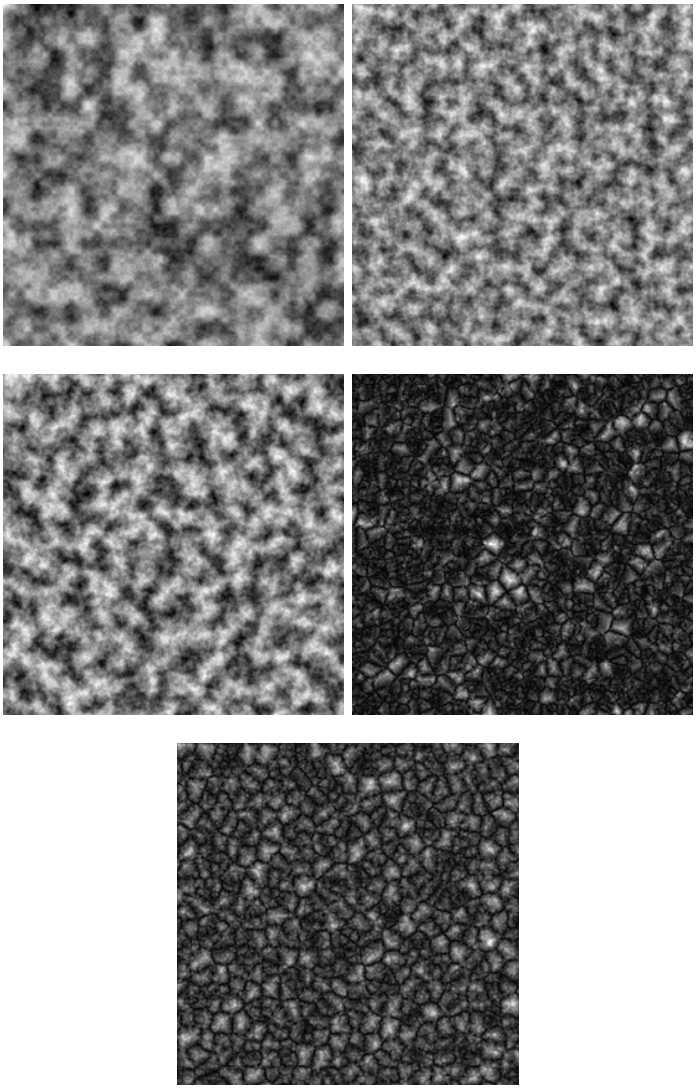

Figure 10: Différents types de bruit[10].
A partir d'en haut à gauche jusqu'en bas à droite : valeur, Perlin, Simplex, Voronoï, Worley.

2.3 Rendu graphique 3D

Il existe deux méthodes principales pour le rendu d'une scène 3D : la rastérisation et le ray tracing.

La rastérisation est un procédé qui consiste à convertir une image vectorielle en une image matricielle destinée à être affichée sur un écran. On englobe dans la rastérisation tous les procédés permettant d'améliorer l'aspect final du rendu en 3 dimensions. Les objets à l'écran sont créés à partir de triangles ou de polygones qui définissent le modèle 3D de l'objet.

- **Avantages :**
  - Faible coût en mémoire,
  - Plus de choix dans la représentation des faces et du modèle (triangles, quadriques, polygones, torres ...),
  - La performance dépend directement du nombre de faces.
- **Inconvénients :**
  - Rendu simplifié non réaliste,
  - Ne prend pas en compte les reflets, la transparence, l'effet miroir.

[10]Image d'une bibliothèque Github par l'utilisateur Scrawk, Accessed January 2020

Une autre méthode existante est le ray tracing qui remédie aux inconvénients de la rastérisation. Cette technique reproduit les phénomènes physiques que sont la réflexion et la réfraction. Cela permet un rendu plus réaliste en tenant compte de la lumière, des reflets ou encore de la transparence. Cependant, il s'agit d'avantages qui ne correspondent pas aux besoins clients. Il a donc été décidé de ne pas se pencher sur cette technique de rendu et d'utiliser la rastérisation.

2.4 Outils existants et prototypes

2.4.1 Blender

Afin de pouvoir générer et afficher la planète, une des solutions qui peut-être abordée est l'utilisation d'outils de modélisation en 3 dimensions. Ils disposent déjà de moteurs de rendu performant, dont certains utilisent un système de scripting et de plugin afin de générer des formes, les modifier, et les texturer, pour ensuite les utiliser plus tard dans d'autres logiciels, selon des formats standardisés.

L'un de ces outils existant est l'outil Blender. Il est tout d'abord un logiciel open source, dont le moteur interne est en C/C++ permettant de bonnes performances. Il dispose aussi d'une API en Python permettant un prototypage très rapide.

Un script python a été écrit permettant de :

1. créer une icosphère d'environ 10 000 polygones
2. appliquer une transformation sur ses sommets selon une texture de Voronoï.
3. afficher les paramètres et le temps de génération.

Le script est disponible dans la hiérarchie du projet *docs/requirements/test_ blender* ainsi que les résultats. Le premier résultat obtenu en exportant l'image depuis l'éditeur est visible sur la figure 11.

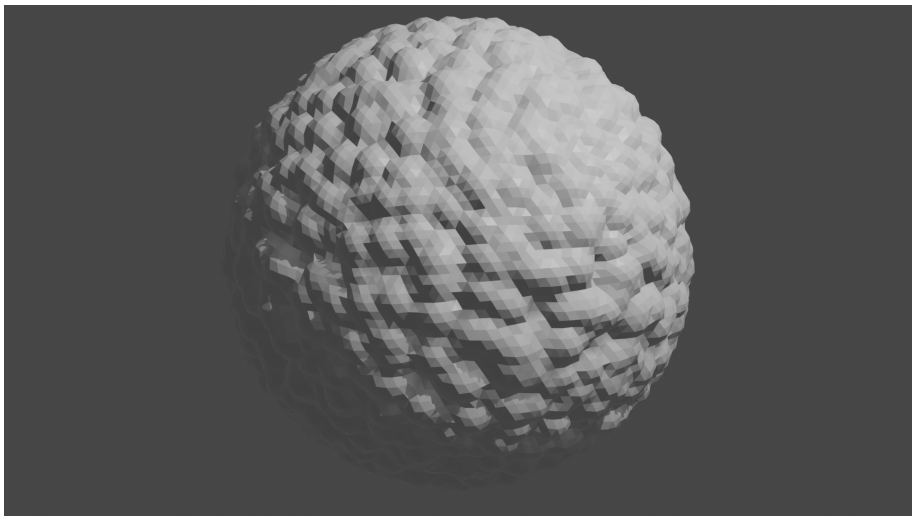

Figure 11: Icosphère générée par Blender. Texture de bruit Voronoï, 6 subdivisions, ~0.02 s, 20480 polygones

Attention à la version de Blender. Le moteur de rendu utilisé est Evee version 2.8 permettant une visualisation sans dégradation lors du déplacement de point de vue (rotation, zoom) avec les rendus finaux. Pour des soucis de performance, il faut aussi faire attention à la machine utilisée, notamment la carte graphique.

Les paramètres ont été changés depuis l'éditeur Blender afin d'arriver aux résultats figure 12 et 13.

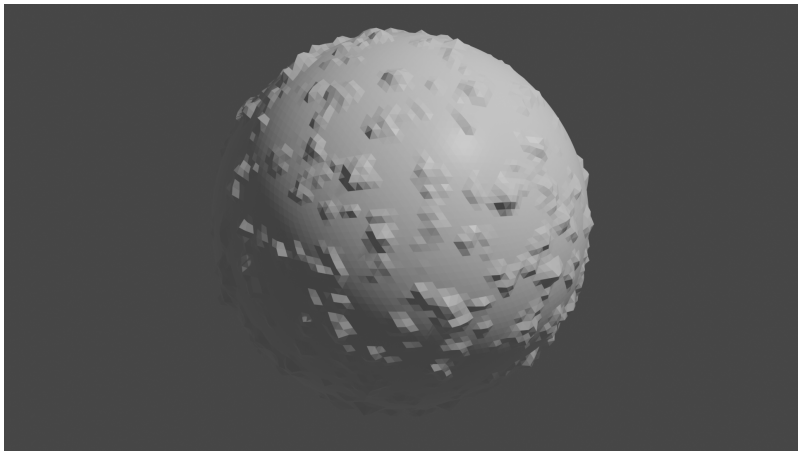

Figure 12: Modification de la texture de bruit Voronoï sans couleur

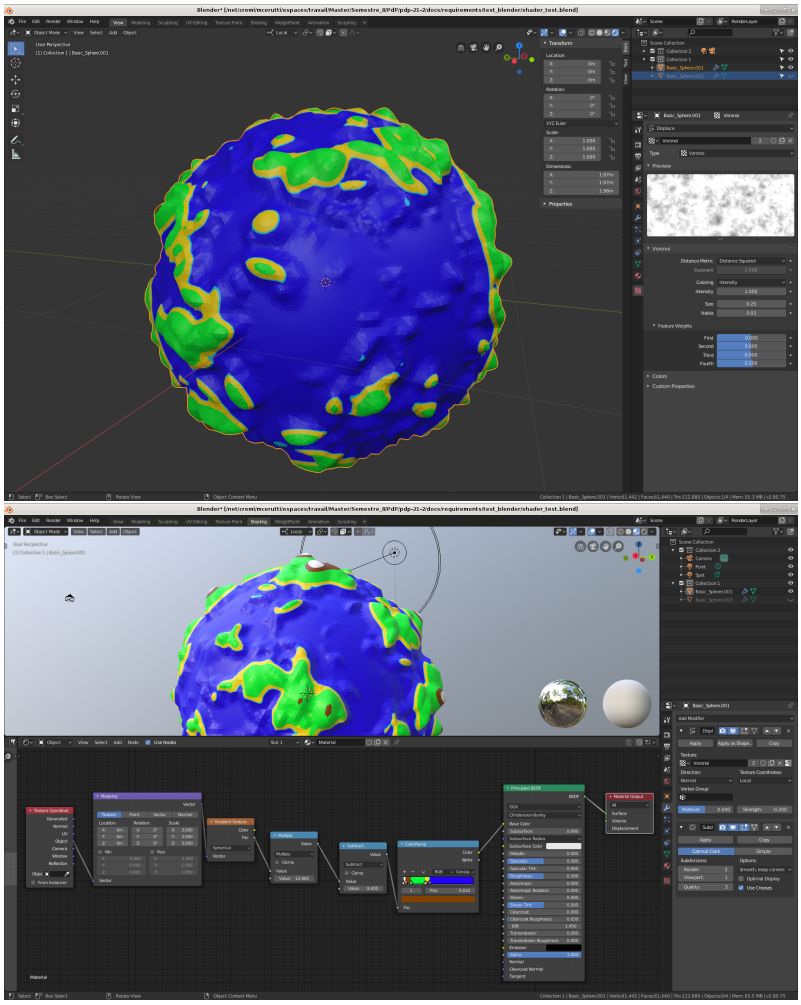

Figure 13: Essais de modifications par l'éditeur Blender : paramètres de texture, bruit de déplacement des sommets, rajout de nuances (shader)

Avantages

1. Rendu de base avec OpenGL, Moteur interne en C/C++ : Permet d'espérer de très bonnes performances au niveau des algorithmes de base. Crucial dans le cas de rendu de l'ordre de 60 images par secondes (FPS) avec une bonne machine.

2. Open source et sous GNU General Public License (ou "logiciel libre") : Cela permettrait de modifier le moteur de rendu, et les algorithmes de visualisation selon les besoins.

3. Multi-plateforme : Permet le portage Linux (correspondant à notre besoin), et aussi sur d'autres plateformes nativement.

4. API très simple en Python afin d'accéder et de modifier la forme : Permet de faire des tests très rapides, et de laisser le client manipuler les algorithmes de modification plus facilement qu'on aurait implémenté en sur-couche du moteur interne.

5. Fonctionnalités supplémentaires (sauvegarde, modification, amélioration de rendu, création de nuauceurs en noeuds...) : Permettrait de se focaliser davantage sur les fonctionnalités concernant la génération de la planète, en ayant déjà les possibilités d'extensions de notre projet.

Inconvénients

1. Ne convient pas à l'exercice du PDP : bien que cet outil puisse répondre à la plupart des besoins de ce projet, il nous contraindrait sur des choix d'architecture et son utilisation différerait de l'intérêt pédagogique de l'unité d'enseignement.

2. Interface de Blender trop riche pour le client : contraire à la demande du client demandant une visualisation simple. Nécessite une connaissance au moins de surface de l'outil. Risque de refus à envisager avant de commencer à l'utiliser.

3. La version de Blender influe sur l'API (la version 2.8 n'est pas encore formellement documentée), et sur les performances de rendu (moteur Evee) : risques d'instabilité et de bug sur l'API fournie, ainsi qu'un développement plus chaotique du fait d'un manque de documentation.

4. Pas de sécurité sur des paramètres trop grands (cesse de fonctionner, pas d'erreurs de ressources si demande trop grande) : lié au fait d'utiliser une API en "boîte noire", nécessite un apprentissage des fonctions du moteur interne et une bonne documentation. Cela aussi influe sur la flexibilité du code et les limites d'extensions avec le logiciel de base.

5. Rendu et génération dépendant de la machine choisi : il est nécessaire d'avoir une configuration suffisamment puissante pour pouvoir utiliser cet outil (se référer à la configuration du client 3.2.3).

2.4.2 OpenGL

OpenGL (Open Graphics Library) est considérée comme une API (Application programming interface) qui offre la capacité d'afficher des objets en 2D et en 3D. Elle est principalement utilisée pour interagir facilement avec les cartes graphiques (GPU). Elle est utilisée en C/C++. Néanmoins, c'est avant tout une spécification, qui décrit ce que sont les entrées et sorties de chaque fonction et comment elles doivent s'exécuter. Cette spécification est implémentée par les constructeurs de carte graphique (dans les pilotes). OpenGL est disponible sous plusieurs versions. La version 3.3 quant à elle, est une base de la programmation moderne d'OpenGL (et donc actuellement la plus utilisée) et permet une grande flexibilité et efficacité. Qui plus est, les versions supérieures à celle-ci implémentent des fonctionnalités supplémentaires sans changer les précédentes. Il y a donc un faible problème de compatibilité. Il est tout à fait possible d'afficher une sphère de plus de 10 000 polygones, comme le prouve la figure 14.

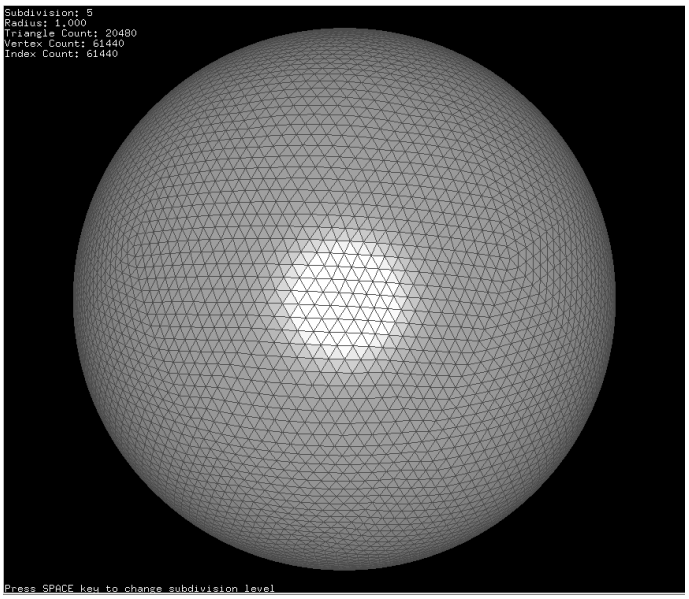

Figure 14: Affichage d'une sphère en OpenGL avec plus de 10000 polygones

Avantages

1. Performant : Ces bibliothèques sont implémentées en C/C++ et sont utilisées par de nombreuses compagnies, studios et équipes de développement afin de faire des applications graphiques performantes (60 images par secondes et définition 4K maintenant). On peut donc espérer avoir de bonnes performances concernant le logiciel.

2. Accessible : Les principales bibliothèques dérivées d'OpenGL ont une bonne documentation et de nombreux tutoriels. Cela permet d'avoir plus de temps pour l'implémentation sachant que l'apprentissage sera plus aisé.

Inconvénient

1. Temps de développement : Du fait que OpenGL est en C/C++, la réfléxion pour la gestion mémoire et l'architecture demandera du temps. Il sera donc nécessaire d'avoir ces faits en tête lors du développement.

2. Complexité de prise en main : OpenGL et ses bibliothèques offrent une très grande flexibilité mais la compréhension et la prise en main de celles-ci est très complexe, comme il y en a beaucoup. Même avec les bons tutoriels et une bonne documentation, il faudra prendre en compte l'apprentissage dans le temps passé au développement.

2.4.3 Gestion de fenêtre et interaction utilisateur

OpenGL ne permet pas directement de gérer des fenêtres et des évènements d'entrée (clavier/-souris). En revanche, il existe principalement deux bibliothèques open source qui étendent OpenGL et permettent cela : freeglut[7] (écrit en C et sous licence X-Consortium) et GLFW[8] (écrit en C et sous licence zlib/libpng).

Ces bibliothèques mettent des fonctions à disposition dont la création de fenêtre en indiquant ses dimensions. Il faut ensuite utiliser une boucle de rendu pour garder la fenêtre ouverte, continuer de dessiner les images et gérer les entrées utilisateur.

Il n'y a pas vraiment de différence de fonctionnalité entre ces deux bibliothèques, si ce n'est que freeglut est basé et étend la bibliothèque GLUT qui était très utilisé par le passé mais sous droit d'auteur privé et dépassé.

2.4.4 Bibliothèques de bruit

Il existe des bibliothèques permettant justement d'implémenter les différents algorithmes vus dans la partie 2.2 dans différents langages. Celles qui ont le plus attirées notre attention sont Libnoise [3] et FastNoise [9] en accord avec les choix précédents.

Libnoise est une bibliothèque sous licence GNU LESSER GENERAL PUBLIC LICENSE Version 2.1 en C++, portable, et qui propose de nombreuses ressources pour la prendre en main. C'est aussi la bibliothèque la plus utilisée à ce jour. Elle permet aussi avec du code utilitaire associé d'exporter les bruits sous forme de textures d'image, et même d'appliquer des traitements plus complexes en combinant les bruits, rajoutant des gradients de couleurs. Un exemple complet est illustré figure 15.

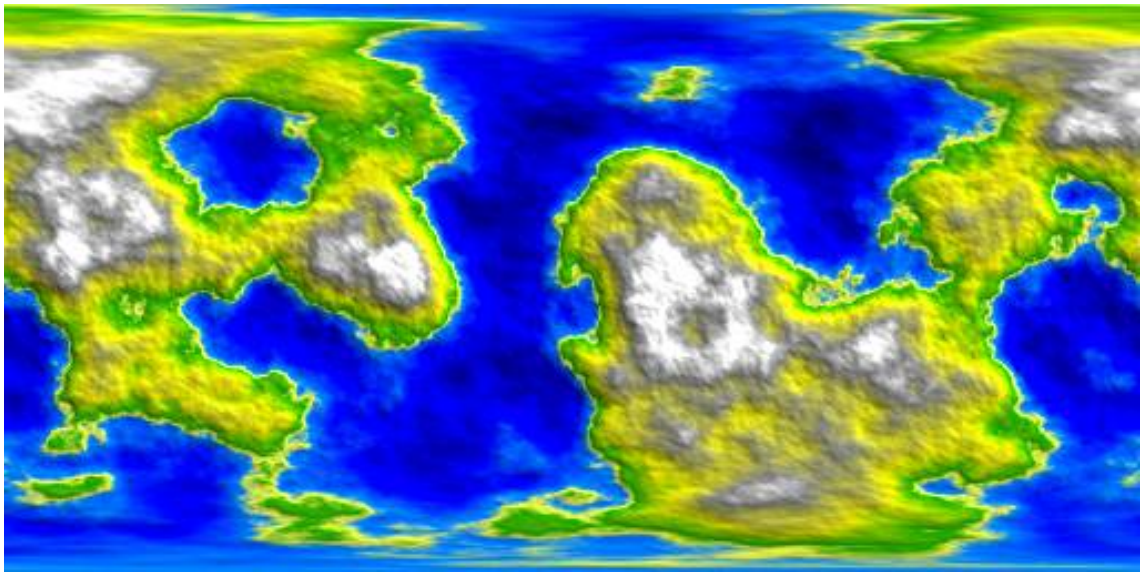

Figure 15: Exemple d'une texture sphérique créée avec libnoise[11].

La bibliothèque FastNoise quant à elle est sous MIT License, et est disponible dans différents langages, C++, C# et aussi sous instructions SIMD. Elle est beaucoup plus récente que la précédente est a notamment de bonnes performances et est plus rapide que libnoise [12].

[11] Image du tutoriel numéro 8, Accessed Marsh 2020

[12] Selon les chiffres donnée par la page : https://jordanpeck.me/2016/05/fastnoise/
Last access : 05/04/2020.

2.4.5 Moteurs de jeux

Il est possible de s'intéresser aux moteurs de jeux. Le seul moteur de jeu qui aurait pu être intéressant est Godot, car il est open source et est sous licence MIT qui a des avantages au niveau de la distribution et de son utilisation comparé aux concurrents les plus utilisés : Unity et Unreal Engine. Mais cette solution a été vite écartée pour trois grandes raisons :

1. La première est que les moteurs de jeux, bien qu'ils aient un moteur de rendu 3D intégré sont beaucoup trop fournis en fonctionnalités. La plupart des fonctionnalités ne sont pas intéressantes pour ce projet.

2. La deuxième est que cela ne convient pas à l'exercice du PDP, en soit faire une sur-couche et faire quelques modifications aurait suffit, mais ce n'est pas le but pédagogique de PDP.

3. La dernière est que bien que le moteur de rendu permet d'avoir déjà des optimisations, il risque d'y avoir beaucoup plus de dépendances et l'installation d'un moteur de jeu par le client, ce qui est contraire à une interface simple.

2.4.6 Cours Université

On s'est intéressé en parallèle du développement au cours de "Mondes 3D" du professeur Pierre BÉNARD en Master 1 Informatique, filière IIS de l'université de Bordeaux. Cela nous a notamment permis d'avoir une base de code avec l'utilisation de l'outil OpenGL et la théorie associée. Avec sa permission, nous avons pu récupérer les ressources[2] pour notre projet.

3 Besoins

3.1 Besoins fonctionnels

3.1.1 Génération d'une planète

3.1.1.1 Géométrie de la planète

La planète est caractérisée par une sphère. Il est donc nécessaire de pouvoir créer une sphère. L'analyse de l'existant a montré qu'il existait plusieurs techniques avec deux approches de départ différentes : avec des points, ou directement avec des formes. La solution la plus adéquate est l'utilisation de l'icosaèdre avec sa subdivision. C'est une méthode tout à fait standard et efficace qui ne nécessitera pas l'implémentation d'algorithmes supplémentaires pour un rendu minimal correct à l'inverse de la sphère de Fibonacci. Qui plus est, l'implémentation et l'utilisation d'algorithmes supplémentaires pour améliorer le rendu est tout à fait envisageable avec l'utilisation de l'icosaèdre. C'est pour cela que nous avons choisi cette solution au vu de l'ensemble des avantages qu'elle présente.

3.1.1.2 Représentation des informations

La géométrie ayant été définit précédemment, une sphère quasi-parfaite sert de base à notre planète. Ce n'est qu'après que l'on peut la modifier à notre guise. Ce qui amène du coup à la définition même des informations que l'on va créer et représenter. Dans notre cas, il a été choisi de ne pas créer d'informations supplémentaires et de se contenter d'utiliser la notion de position et de couleurs offertes par les points. La planète ne se voulant pas être une simulation réaliste, il n'a pas été nécessaire de mettre en place des pôles, l'équateur, des biômes et une logique scientifique vis à vis de la position des reliefs (plaques tectoniques). L'ajout de ces informations reste cependant une amélioration possible.

3.1.1.3 Paramètres modifiables

Les informations générées doivent en partie être contrôlées par l'utilisateur. S'il le souhaite, il doit pouvoir fixer certains paramètres de la génération, ou au contraire, les laissés aléatoire avec un moyen de contrôle sur eux. Ces paramètres pourraient être changer à chaque exécution du logiciel sans recompilation, mais pas dynamiquement une fois que la planète est crée. Nous avons choisi pour cela de faire un système de fichier de configuration avec un parseur en XML grâce à la bibliothèque pugyXML, afin de pouvoir lire et changer les paramètres à l'exécution.

3.1.1.4 Gestion de l'aléatoire

Comme énoncé précédemment, certains paramètres pourraient être gérés aléatoirement, mais il doit être possible de recréer la même planète, ou au moins une dont les caractéristiques se rapprochent. Il faut donc communiquer aussi ces informations à l'utilisateur pour qu'ils puissent les modifier s'il le souhaite. Pour des paramètres simplement aléatoire de type "bruit blanc" (random), la bibliothèque standard suffit, pareil pour des lois de distribution classique comme la distribution gaussienne, où l'on peut afficher cependant l'information de la graine aléatoire (seed). Pour de l'aléatoire plus réaliste, la bibliothèque Libnoise choisie permet justement de générer des bruits avec les propriétés qui nous intéressent.

3.1.1.5 Affichage de la planète

Une seule planète doit être affichée à la fois. Afin d'y parvenir, il fallait donc créer une fenêtre (éventuellement extensible) qui se lancera lors de l'appel à l'exécutable. Étant donné la visée ludique de cet outil, une fenêtre carrée ou rectangulaire avec seulement la planète au centre sembla être le plus appropriée pour l'immersion et l'interaction avec la souris. Pour ce faire, nous avons préféré la bibliothèque open source GLFW qui permet de gérer une fenêtre, un contexte, des évènements clavier/souris etc. Afin aussi de pouvoir afficher la planète, on utilise OpenGL avec les biblothèque Eigen et Glbinding pour facciliter la manipulation des données, ce qui nous permet d'exploiter la puissance de la carte graphique.

3.1.1.6 Interaction utilisateur

Il fallait permettre à l'utilisateur de visualiser la planète sous différents angles. Pour cela, il devait être possible de tourner et de se déplacer précisément autour de la planète. Un point de vue a été défini par une "caméra" qui se déplace lors d'un mouvement de la souris lorsque le clic gauche est enfoncé. Cette caméra a comme point d'encrage le centre de la planète. Elle effectue donc une rotation orbitale autour de celle-ci. Tout cela est rendu possible par l'intermédiaire des bibliothèques open source Eigen (pour les mouvements vectoriels) et GLFW (pour la fenêtre et les évènements clavier/souris) (pour plus de détails, voir [4]).

3.1.2 Sauvegarde/Chargement d'une planète

L'utilisateur devait avoir la capacité de sauvegarder une planète afin de la réutiliser dans un autre logiciel. Il doit être aussi possible de recréer une planète similaire à une autre à partir des mêmes paramètres de génération. Nous avons choisi pour l'exportation les formats .obj et .off, en partant du code de [2] et pour la recréation voir les besoins 3.1.1.3 et 3.1.1.4.

3.2 Besoins non fonctionnels

3.2.1 Ergonomie d'utilisation

Afin de laisser plus de liberté à l'utilisateur, on a pensé à des options lors de l'exécution du programme mais comme il y a beaucoup de facteurs qui rentrent en jeu, il est fastidieux de tout spécifier en ligne de commande, nous proposons alors des fichiers XML que l'utilisateur pourra modifier afin de personnaliser au plus possible les paramètres de la planète qu'il souhaite générer (le niveau de subdivision, les seuils sur les altitudes, la couleur en arrière-plan ...). GLFW en plus de donner la possibilité d'avoir une fenêtre pour afficher notre contexte OpenGL est également un outil qui permet la gestion des interactions clavier entre l'utilisateur et la fenêtre. On peut donc assigner à des touches toutes les petites modifications afin que l'utilisateur ait le choix d'afficher ou non les options qu'on a implémenté (activer/désactiver la lumière par exemple). Les interactions souris peuvent également être gérées via cette bibliothèque ce qui permet d'assigner les options de navigation sur la planète (rotation/zoom) uniquement à l'aide de la souris de manière assez instinctif.

3.2.2 Rapidité d'affichage

Il nous faut différencier et définir les différents temps qui s'appliquent à ce projet.

Le temps de génération désigne le temps nécessaire à la génération procédurale de la planète (géométrie et informations générées procéduralement). Il doit être de l'ordre de quelques secondes seulement (moins de 5 secondes environ).

Le temps d'affichage désigne le temps nécessaire à l'affichage de la planète dans la fenêtre. Il doit être de l'ordre de la seconde.

Le temps de rafraîchissement désigne le temps entre deux affichages après un mouvement de la caméra. Il doit être de l'ordre de quelque dixièmes ou centièmes de secondes (30 images par seconde) pour avoir une ergonomie de visualisation correcte.

3.2.3 Compatibilité

Le client doit pouvoir l'utiliser sur sa machine, et il faut donc s'adapter à son matériel, n'ayant pas forcément comme machine une configuration du CREMI, qui est celle que nous utilisons pour développer le logiciel.

Cela induit une compatibilité avec **Gentoo Linux** et un test de performance sur une configuration proche de la machine visée. La configuration étant la suivante : Processeur Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, une carte graphique NVIDIA Corporation GF108GL [Quadro 600] ainsi que 16376040 kB de RAM. Tous les outils envisagés ont des performances variables suivant la machine visée, c'est donc un **point critique** à ce projet.

3.2.4 Rendu esthétique

Il faut un certain nombre de polygones pour rendre une visualisation de la planète assez détaillée pour le client (environ 10000 après discussion avec celui-ci). Ce besoin influe de manière importante sur les temps de génération et d'affichage, ainsi que sur l'aspect visuel de la planète.
Il nous faut aussi respecter le style graphique dit "Low-Poly", ce qui indique un éclairage pas forcément réaliste mais cohérent avec de l'éclairage et des ombres. Ce besoin s'est aussi précisé à la demande du client pour pouvoir avoir une surface de l'eau transparente et dont on peut personnaliser la couleur.

3.2.5 Maintenabilité

L'architecture a été conçue afin de permettre à un autre développeur de pouvoir apporter des changements ou corrections de manière assez aisée, par exemple s'il veut pouvoir changer la forme de la planète et/ou apporter sa propre méthode de subdivision, c'est rendu possible en créant une classe héritant de shape. Il peut également apporter un autre éditeur, du moment qu'il hérite de la classe Editor.

4 Architecture du projet

Pour l'architecture nous nous sommes basés sur le code du cours de Mondes 3D pour les Master IIS, plus précisément le TD9, par Pierre BENARD [2]. L'architecture se décompose alors en deux parties: Une partie appelée model qui contient l'ensemble des informations à manipuler (principalement la géométrie). L'autre partie s'occupe quant à elle de la visualisation de ces informations. **L'architecture complète de notre logiciel est disponible en annexe** (voir **8 Annexe**).

4.1 Modèle

Le modèle correspond à la partie qui s'occupe de générer une planète : créer une forme, lui appliquer des reliefs, et des couleurs. La représentation de la planète est faite par l'interface **Shape** et la classe qui l'implémente, **Icosphère**. La partie qui s'occupe de la modification de ces informations est représentée par l'interface **Editor** et les deux éditeurs qui l'implémentent : **Random_Editor** et **Noisyheight_Editor**. La figure 16 démontre une version simplifiée de la partie model.

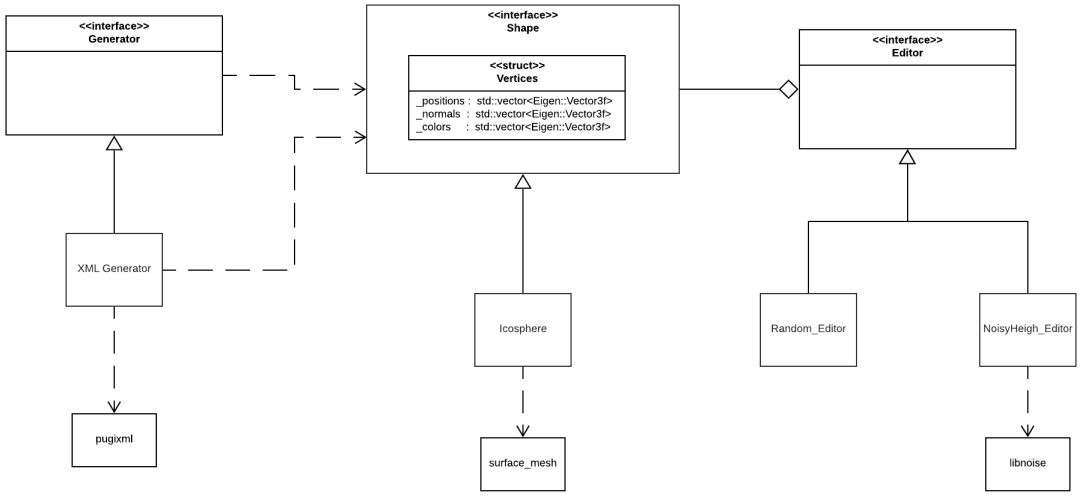

Figure 16: Diagramme simplifié du package model

4.1.1 La représentation d'une planète

Afin de laissez le choix au développeur d'implémenter une planète de la manière qu'il le souhaite, le choix a été fait de faire de **Shape** une interface assez légère (peu de contraintes au niveau de la manipulation des informations contenues dans la structure Vertices). Cette interface est représentée par la figure 17. Elle contient notamment la structure **Vertices**, utilisée pour stocker l'ensemble des points, leurs normales et les couleurs associés pour permettre la visualisation. Le choix est laissé au développeur d'utiliser la structure de données qu'il veut pour gérer la connexité des points. Ainsi, nous limitons la dépendance à une bibliothèque en particulier (ici surface_mesh) pour gérer cette connexité.
Dans ce projet, c'est une icosphère (icosphere.cpp) qui a été choisie pour représenter une planète sphérique. Pour gérer la connexité des points et des faces, nous avons utilisé la bibliothèque surface_mesh. En effet, comme nous sommes partis du TD9 de Pierre Benard sur les mondes3D [2], une simplification de la bibliothèque surface_mesh était proposée. Cette bibliothèque permet principalement de savoir les voisins de chaque point constituant la planète, ainsi que les faces qui sont autour d'un point. Il est tout à fait possible de rajouter des propriétés à chaque point par l'intermédiaire de cette bibliothèque. Néanmoins, une fois qu'une valeur est ajoutée à une propriété, il est impossible de la changer à moins de refaire complètement cette structure de données. Ainsi,

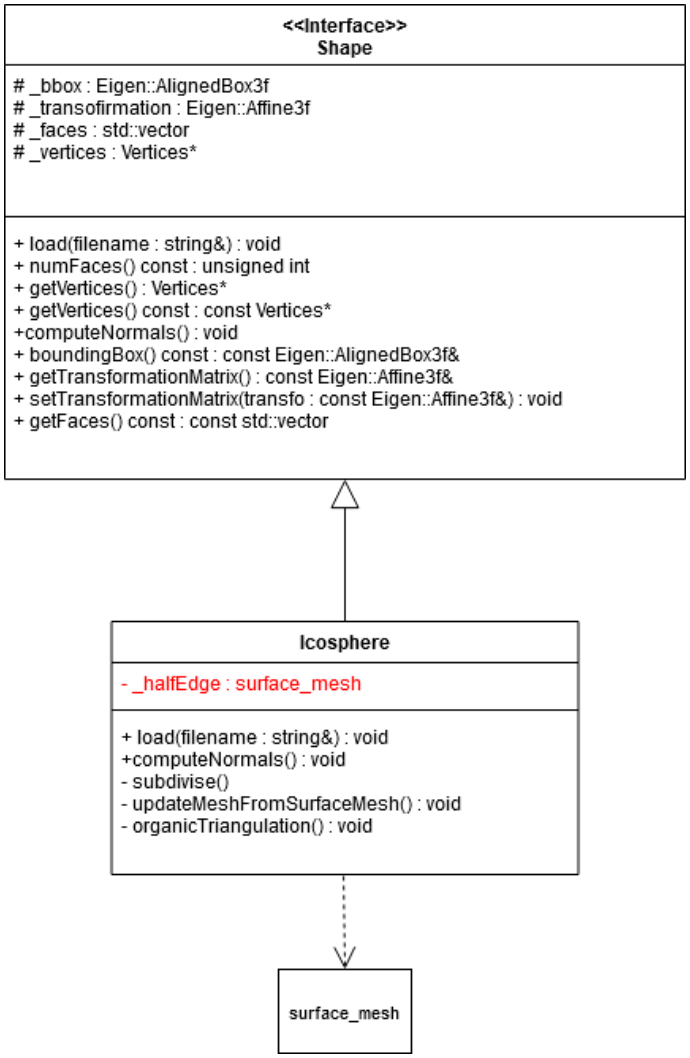

Figure 17: Diagramme de la classe représentant une planète

cela a motivé le choix de faire une deuxième structure qui est **Vertices** pour stocker les coordonnées, les couleurs et les normales et garder celle de surface_mesh uniquement pour la connexité.

4.1.2 Modification d'une planète par un éditeur

Il a été décidé de faire une dissociation entre la représentation d'une planète et sa modification. La raison est d'avoir la capacité de créer plusieurs éditeurs de planètes et de pouvoir les appliquer à la même planète de manière cohérente. L'éditeur représente donc cette capacité à modifier la représentation d'une planète et produit la planète finale. Cela évite d'autant plus d'avoir un objet beaucoup trop complexe. Une seule méthode doit être implémenter : edit. C'est cette dernière qui s'occupera de modifier l'ensemble de la planète.
Nous mettons à disposition deux éditeurs qui implémentent cette interface comme dit précédemment : **Noisyheight_Editor** qui produit un bruit de perlin afin de changer le relief et **Random_Editor** qui lui est complètement aléatoire. Ces deux éditeurs prennent la hauteur des points résultantes afin de leur attribuer une couleur.
Une autre conséquence positive de cette séparation est que la forme de départ, l'implémentation de l'interface **Shape**, peut être ré-implémentée par un autre algorithme que celui que l'on a choisi (3.1.1.1). Ainsi on pourrait remplacé l'icosphère par un autre algorithme (voir 2.1).

4.1.3 Paramètres modifiables par l'utilisateur

Générateur XML : Afin de répondre au besoin de modification des paramètres (3.1.1.3), nous avons mis en place une interface **Generator** dont sa responsabilité est de lire un fichier de configuration, et de créer à la volée la forme de base et appliquer l'éditeur renseigné avec les différents paramètres. Il retourne ensuite la planète créée qui peut alors être utilisée par la partie visualisation.
Cela a permis notamment de faciliter les modifications appliquées par l'utilisateur, pouvant alors librement choisir les paramètres qu'il veut, jusqu'à choisir les paramètres aléatoirement avec l'énumération **AleatoryMode** en paramètre.

L'implémention actuellement de cette interface se fait avec la bibliothèque pugixml, avec les fichiers de configuration associés (figure 18).

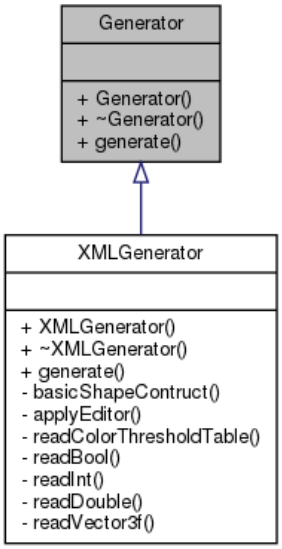

Figure 18: Diagramme de classe simplifié de Generator et ses implémentations (doxygen).

On a gardé cependant la possibilité dans le main de changer le comportement par défaut si il n'y a pas de fichier de configuration.

Couleurs paramétrables : Lié au besoin de l'implémentation précédente, il a fallu créé un système qui puisse déterminer sur une échelle de hauteur les couleurs à utiliser dans les éditeurs. C'est le rôle de la classe **TresholdTable**, qui permet, selon une hauteur de sommets (par convention ramenée dans le domaine [-1,1]), d'associer une couleur selon des bornes (**Treshold**) prédéfinis, illustré figure 19. Comme c'est une classe template elle serait extensible à d'autres problèmes. On pourrait changer par exemple la représentation des couleurs ou encore pouvoir, sur une échelle de valeurs d'un paramètre, découper son domaine et associer à chaque borne une donnée.

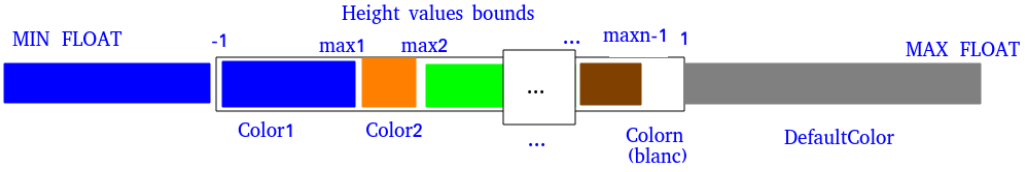

Figure 19: Principe illustré avec des couleurs de la classe TresholdTable.

4.1.4 Aléatoire et bruit

Afin d'avoir une sorte de boite à outil pour tout ce qui concerne l'aléatoire et les bruits, nous avons défini le namespace **NoiseRandom**. Il permet de centraliser les fonctions globales, facilitant la manipulation de l'aléatoire, mais aussi, grâce à la classe **HeightNoise** au sein de ce namespace, de fournir une abstraction de la bibliothèque Libnoise et de séparer cette dépendance. Dans le cas où l'on utiliserait une autre bibliothèque, on aurait, en changeant l'implémentation de HeightNoise, un impact minimal sur le reste du code si on ne change pas les signatures de fonction.

4.2 Visualisation

La visualisation regroupe l'ensemble des classes permettant l'affichage et l'interaction avec l'utilisateur. L'ensemble de ces classes se trouve dans le répertoire view. L'affichage devant se faire sur un système d'exploitation Linux, il a donc été choisi de le faire via OpenGL. Pour permettre toute extension (via Windows avec DirectX par exemple), on a utilisé le pattern bridge via l'interface Rendering. Cette dernière est implémentée par Rendering_OpenGL. Toute fonction de visualisation de la planète passe donc par cette interface. Elle est principalement utilisée dans la classe **Viewer**, qui s'occupe à la fois de générer une visualisation mais aussi des interactions avec l'utilisateur (déplacement de la caméra et zoom principalement). Ces interactions sont gérées ici par la bibliothèque GLFW. La capacité de déplacement de l'affichage est faite par l'intermédiaire de la classe caméra qui hérite de trackball, cette dernière contenant l'ensemble des méthodes nécessaires au zoom et à la rotation, ainsi qu'aux matrices de projections.

4.3 Récapitulatif des dépendances

Mozilla Public License 2 :

- Eigen : facilite les calculs matriciels pour la géométrie.
- glbinding : permet le lien entre les données en mémoire et OpenGL.
- glfw : API utilisée pour la gestion de fenêtres et les intéractions clavier/souris.

GNU Library General Public License, Version 2 :

- surface_mesh : facilite la gestion de connectivité et connexité des vertices pour la subdivision. Copyright (C) 2013 by Graphics & Geometry Group, Bielefeld University
- Libnoise : utilisée pour créer les reliefs à partir de bruits cohérents.

MIT License :

- pugixml: permet la prise en charge de fichiers de configuration XML.

Copyright 2008, Google Inc. :

- Googletest : permet de faciliter la mise en place de tests unitaires automatisés.

5 Implémentation

5.1 Modèle

La figure 20 représente le diagramme de séquence de la partie modèle. Il commence par la lecture du fichier de configuration par l'objet XMLGenerator qui va, le long de la lecture, renseigné tous les paramètres pour d'abord créer la géométrie de base, ici une icosphère, et ensuite la modifier avec un éditeur, par exemple NoisyHeight_Editor. Cas spécial, si le paramètre n'existe pas dans le fichier pour l'éditeur on choisira le paramètre aléatoirement grâce au renseignement par un système de "flags" dans le constructeur, Icosphere n'ayant que des valeurs par défaut dans ce cas là, il n'en a pas besoin. A la fin de la génération, on donnera la planète obtenue à la partie visualisation avec au passage l'affichage des paramètres.

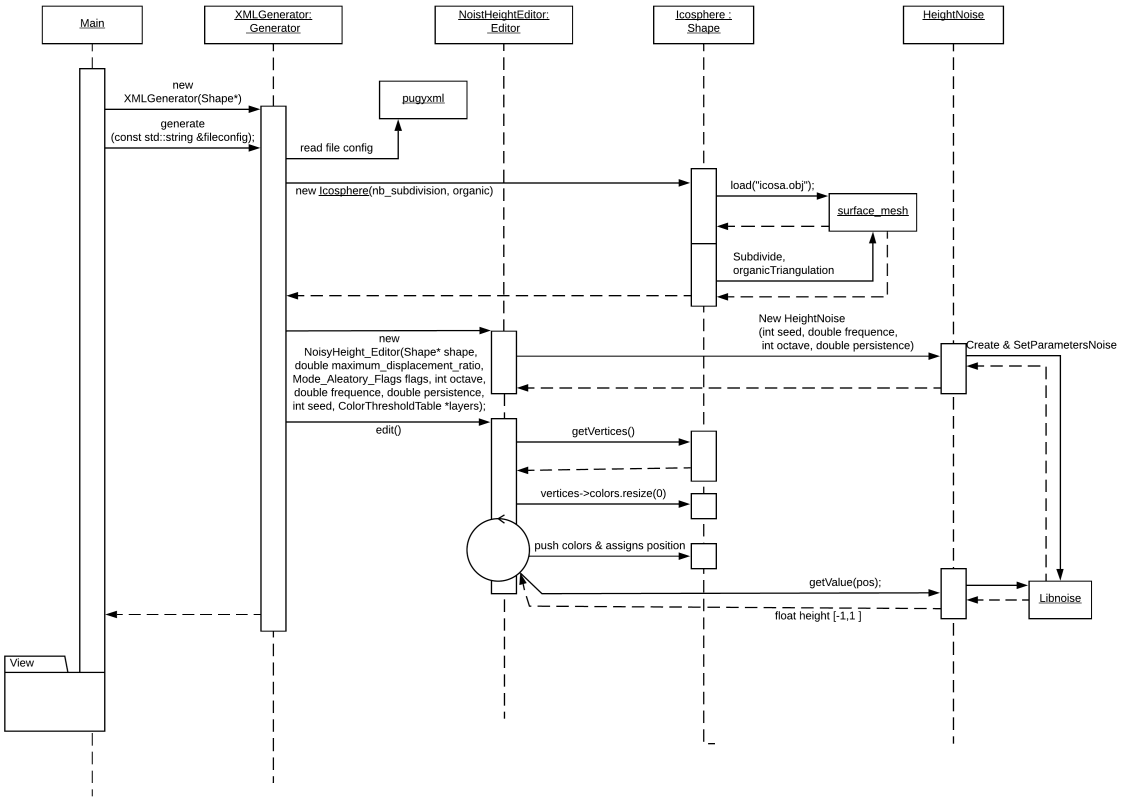

Figure 20: Diagramme de séquence de la partie model.

5.2 Visualisation

On crée ensuite une fenêtre OpenGL grâce à GLFW, et on envoie les informations géométriques et graphiques de la planète au GPU pour manipuler la représentation via le vertex shader et le fragment shader (figure 21)(renseigné dans src/data/shaders/) afin de permettre l'affichage dans cette même fenêtre. Le vertex shader a pour rôle de manipuler les coordonées 3D de chaque sommet et est appelé une fois par sommet. Le fragment shader redéfinit les coordonées RGBA (rouge,vert,bleu pour la couleur et alpha pour la transparence) pour chaque pixel traité, ce shader est appelé une fois par pixel. Les shaders reçoivent donc les informations pour chaque sommet (position, normale, couleur) et les interprètent afin de savoir quelle couleur afficher dans la fenêtre, en utilisant la théorie d'éclairage de Blinn-phong[13]qui permet d'ajouter un effet d'ombre pour chaque point en fonction d'une source de lumière dans la scène 3D [10]. L'affichage est ensuite constamment actualisée, afin de permettre l'interaction avec l'utilisateur.
Pour les mers et les océans, nous utilisons les mêmes shaders mais en mode "océan", afin de ne

pas changer ou copier les coordonnées pour la surface transparente de l'eau. Autrement dit, les données sont envoyées une fois du CPU vers le GPU, puis sont utilisées 2 fois par le GPU. En effet, on fait l'affichage en 2 passes : une première passe pour la planète modifiée sans les océans, et une seconde passe via ce mode pour que le GPU aplatisse le relief, et applique une couleur prédéfinie dans les shaders. En procédant ainsi, nous réduisons les transferts entre CPU et GPU, et gagnons donc en mémoire et en temps de calcul.

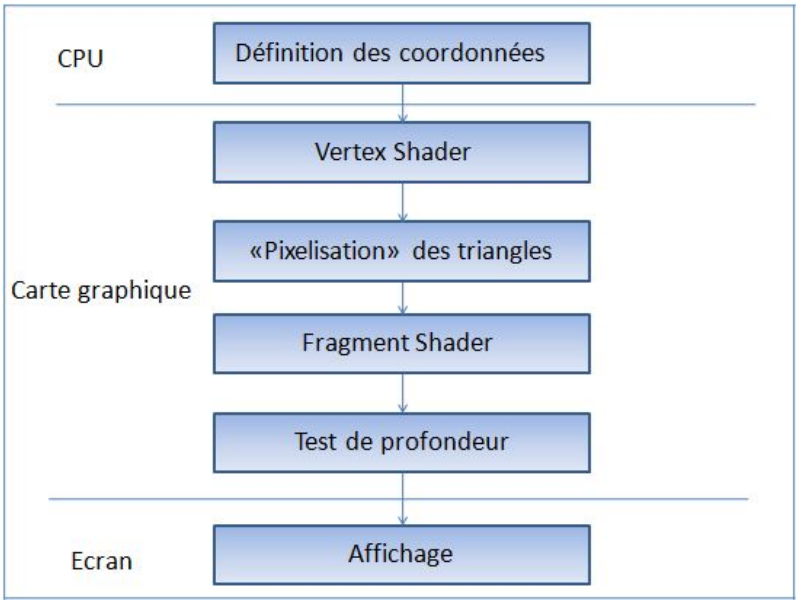

Figure 21: Pipeline d'affichage graphique 3D OpenGL [14].

La figure 22 montre le diagramme de séquence de la partie visualisation. On y voit surtout du coup les différentes étapes avec tout d'abord l'initialisation de la fenêtre par GLFW, puis ensuite les différents composants **Rendering_OpenGl**, les shaders, la caméra, pour ensuite durant la boucle de rafraîchissement et d'interactions, actualiser les états de OpenGL, avec en dernier la fonction draw() de **Viewer** qui s'occupe en dernier d'envoyer les informations au GPU.

[13]https://en.wikipedia.org/wiki/Blinn–Phong_reflection_model, Accessed 4 April 2020
[14]https://openclassrooms.com/fr/courses/966823-developpez-vos-applications-3d-avec-opengl-3-3/961139-introduction-aux-shaders, Accessed 3 March 2020

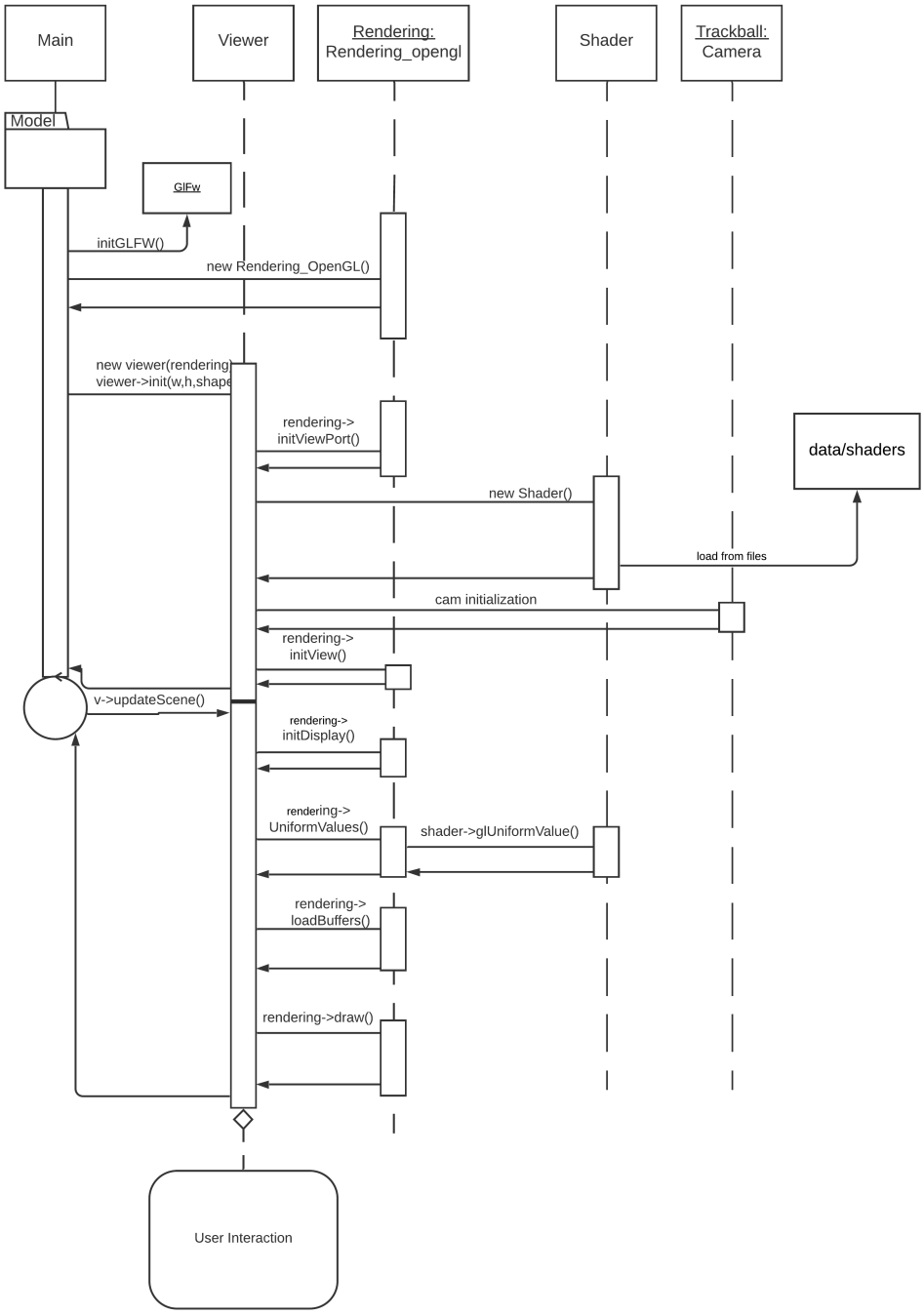

Figure 22: Diagramme de séquence de la partie visualisation.

A tout moment, l'utilisateur peut faire tourner la planète à l'aide de la souris (en cliquant et glissant dans la fenêtre), et effectuer un zoom/dézoom sur la planète. Il peut également appuyer sur S pour sauvegarder et W pour afficher le maillage en mode fil de fer. L permet d'activer/désactiver la rotation de la lumière pour un pseudo cycle jour/nuit. Ces interactions sont détaillées dans la figure 23.

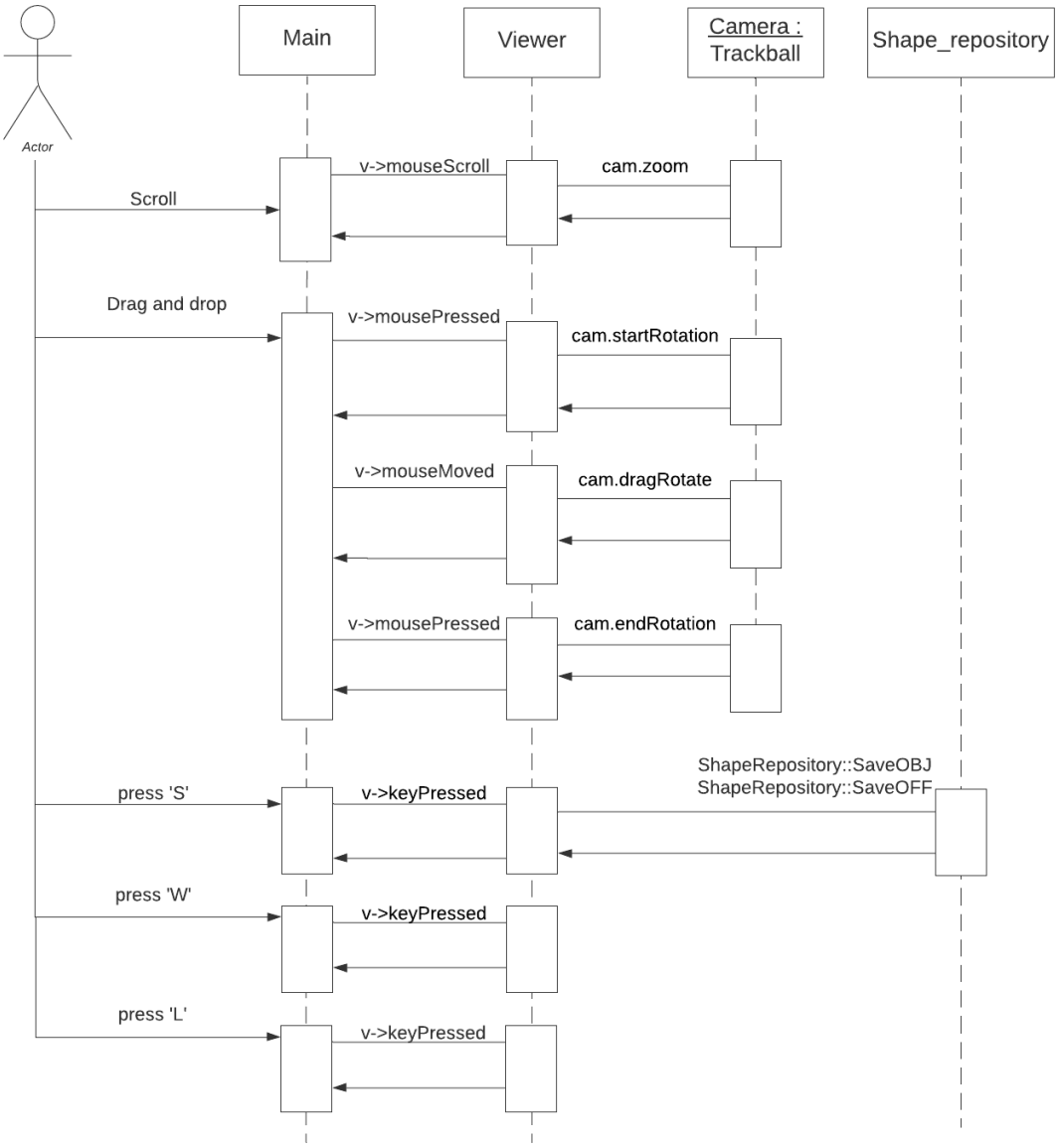

Figure 23: Diagramme de séquence de la partie interaction.

Après exécution de notre logiciel avec le fichier de configuration "simple.xml", on obtient une planète comme la figure 24. On peut lire ensuite dans le terminal les différents paramètres qui ont été utilisés afin de créer un fichier de configuration avec les mêmes paramètres pour pouvoir la recréer et faire de légers ajustements.

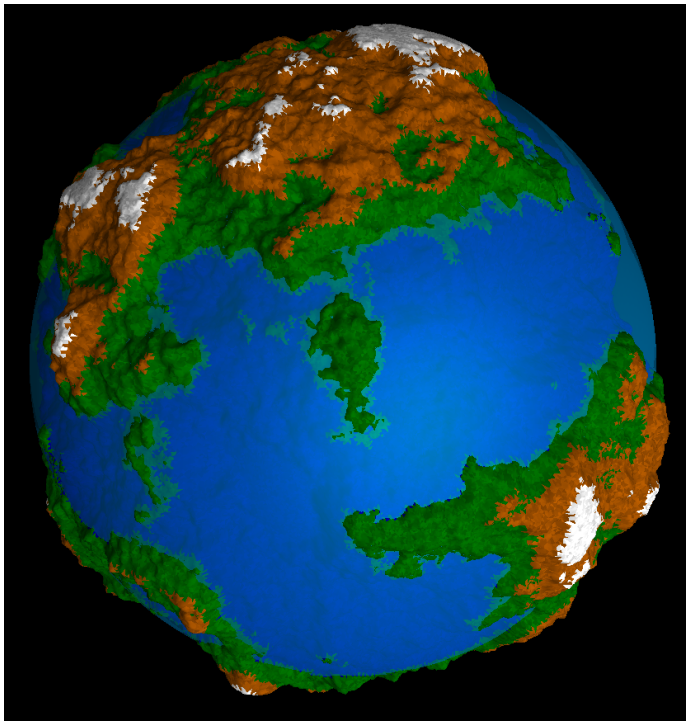

Figure 24: Une planète générée à partir de "simple.xml"

Un autre exemple de planète plus "martienne" à la demande du client obtenable avec notre logiciel est présenté figure 25. Il a suffit de changer la couleur de l'océan dans le shader "blin.vert", mais il pourrait être rajouté en paramètre du viewer. Les paramètres de génération sont disponible dans le fichier "src/data/generator/client.xml"

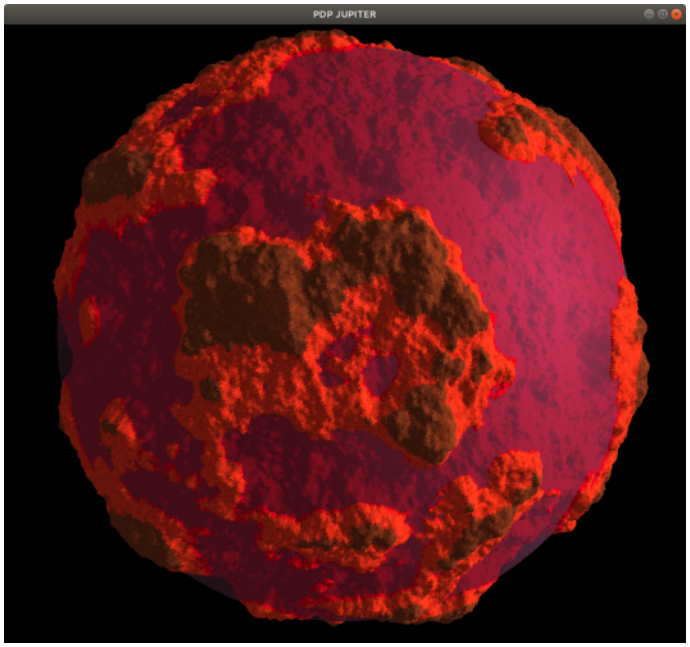

Figure 25: Une planète dans un style martien

6 Tests

Pour les tests nous avons utilisé activement le framework GoogleTest, notamment pour les tests unitaires sur la partie génération. Pour la partie visualisation, comme elle concerne surtout l'esthétique c'est principalement des tests qualitatifs par l'équipe de développement et le client qui ont été faits.

6.1 Tests esthétiques et interactions

Afin de vérifier l'aspect esthétique et l'interaction de la solution, nous avons fait des tests qualitatifs au sein de l'équipe, mais nous avons aussi convié deux fois le client durant la période de développement à nous faire des retours sur ses impressions de la solution.
Au premier retour, cela a permis notamment de privilégier la partie visualisation avec l'éclairage et l'amélioration du rendu de l'eau pour qu'elle soit transparente.
Au deuxième, cela a permis de personnaliser certaines touches suivant ses préférences, et de mettre aussi en évidence les points à améliorer au niveau de l'utilisation et la documentation du logiciel.

6.2 Tests sur les éditeurs

Le test sur les éditeurs concerne principalement le seul éditeur paramétrable et où l'aléatoire n'entre pas trop en compte. En effet, il est difficile (et peu utile) de tester si un éditeur aléatoire fonctionne correctement (cas du random_editor). Dans le cas du noisyheight_editor, il est possible de vérifier qu'il fonctionne correctement par le domaine des informations de sortie.

Le but du test est de vérifier à paramètres équivalents que la génération doit être identique au niveau de la position des points et des normales (ce qui n'est pas le cas des couleurs qui peuvent légèrement varier au vu de la palette disponible). Pour se faire, nous procédons à deux générations de planète avec les mêmes paramètres. Pour cela, nous avons fixé à 10 les paramètres d'octave, de fréquence et de persistance. Nous comparons la valeur de chaque point dans le tableau de positions. Nous faisons la même chose pour les normales entre les deux planètes. Si les deux sont parfaitement identique, les deux planètes générées avec les mêmes paramètres sont identiques. Le test est donc validé. Si elles ne sont pas équivalentes, cela lève une erreur et le test ne passe pas.

6.3 Tests sur les performances

On cherche ici à évaluer les performances de calcul en fonction du nombre de subdivisions et de l'application de notre éditeur. Pour cela on répète la même subdivision 50 fois afin d'avoir une moyenne par valeur de subdivisions, et on augmente par la suite le nombre de subdivisions. On peut alors savoir jusqu'où il est possible d'aller dans le nombre de subdivisions si on ne donne pas de bornes supérieures. On compare aussi le temps de calcul avec et sans éditeur, en fonction du nombre de subdivisions et nous obtenons alors les graphique visible à la figure 26 et 27.

On remarque alors que passé 10 subdivisions, le temps commence à augmenter de manière exponentielle. C'est bien ce que l'on attendait vu que l'on quadruple le nombre de faces à chaque subdivision, ce qui revient à une complexité exponentielle comme observé. Même si un temps de 10 secondes reste tout à fait raisonnable, si l'on va plus loin, le test nous a permis de mettre en évidence un manque de mémoire afin de réaliser la génération qui se solde par une exception de "bad alloc". Ce problème mémoire est notamment lié au nombre de points générés à chaque subdivision. Pour chaque triangle, face élémentaire de notre icosphère, nous créons 4 nouveaux triangles, ce qui augmente à chaque fois le nombre de points de 3 pour chaque triangle.

Un exemple du nombre de sommets :

Listing 1: Résultat des tests unitaires

```
Maximum subdivision with NoisyHeight_Editor : 10 for 2835 ms.
Number of vertices : 2621442
```

Cependant, ce résultat n'est pas contraignant pour ce projet puisque nous souhaitons des rendus "low-poly", c'est à dire avec peu de polygones. Nous pouvons quand même faire des rendus visuellement agréables sans pour autant avoir beaucoup de polygones. La barre des 10 000 est

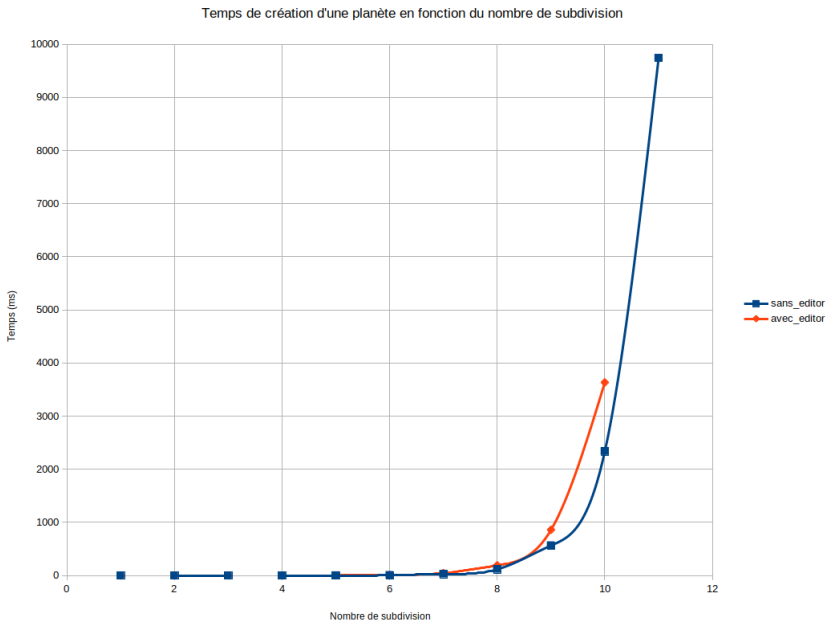

Figure 26: Graphique représentant le temps de génération en fonction du nombre de subdivisions et l'application ou non d'un éditeur

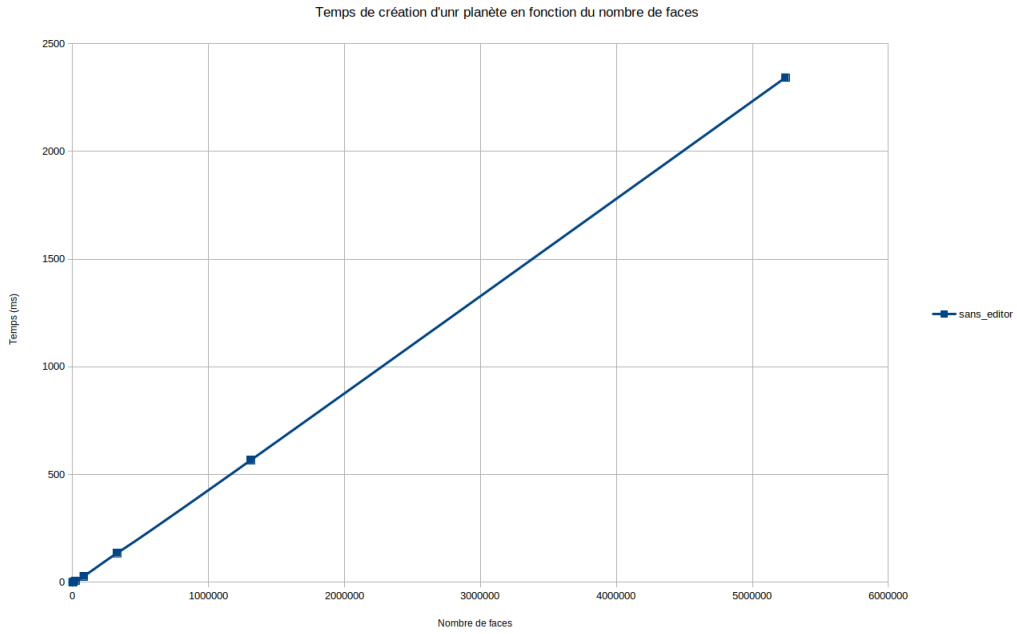

Figure 27: Graphique représentant le temps de génération en fonction du nombre de faces (correspondant à un niveau de subdivision).

dépassée sans trop de difficultés, donc la complexité n'est pas critique pour l'objectif de ce projet.

On a réalisé un autre test plus global portant sur le temps de génération d'une planète. Le but de ce test est de déterminer le nombre de subdivisions possible dans un certain temps donné. Ainsi, on cherche à obtenir avec ce test quel est le nombre de subdivisions idéal dans un temps arbitraire (que l'on peut voir comme une contrainte du client). On part donc d'une subdivision à 0, et on augmente tant que le temps ne dépasse pas celui que l'on souhaite avoir. Dans le cas de notre test, on prend en compte le temps de modification de la planète en lui appliquant le NoiyHeight_Editor.

6.4 Test sur la sauvegarde

Pour que la sauvegarde fonctionne correctement, il faut s'assurer que les données sauvegardées soient identiques à la planète qui a été générée. Pour se faire, nous mettons en place un test scénario. Nous sauvegardons les données au sein d'un fichier .obj que nous lançons sur un logiciel de visionnage (type blender). Si le fichier n'est pas correct, il est facile de l'observer car la planète générée ne sera pas du tout la même. Il est tout à fait possible de sauvegarder la planète 2 fois et vérifier que les données soient les mêmes, mais ce genre de test ne garantie pas que les données sauvegardées soient les bonnes. La seul référence possible est une référence visuel. Nous pouvons tout à fait lancer le fichier depuis notre exécutable pour comparer que les deux planètes sont identiques (il n'est donc pas forcément nécessaire d'utiliser un logiciel tiers pour comparer).

6.5 Tests de non régression

- Coordonnées : On a vérifié le bon passage des coordonnées cartésiennes à sphériques, et vice versa, tantôt avec des valeurs d'oracle, tantôt avec des valeurs aléatoires où on doit pouvoir revenir à la cordonnée de départ. On a aussi vérifié certains points particulier comme les points (0,0,0) en coordonnées cartésienne et (0,x,y) en coordonnées sphériques, correspondant à une sphère de rayon nulle, et donc une infinité de point en coordonnées sphériques.

- Random : On a vérifié que les fonctions de hasard suivent bien le comportement indiqué, autant sur les domaines que sur les lois qu'elles doivent suivre selon la loi des grands nombres.

- Treshold : On a vérifié la bonne construction et l'ordre des couches dans la table ainsi que la non-modification des données durant le processus. On a aussi testé si, sur une valeur comprise dans l'intervalle du minimum et du maximum des bornes de la table, la fonction getColorDataByValue renvoie bien la bonne donnée de la couche à laquelle cette valeur appartient. Comme le nombre de couches et les données ne sont pas importantes pour ces tests, c'est décidé aléatoirement.

6.6 Tests mémoires

Nous nous sommes occupés des fuites mémoires avec l'outil Valgrind, à la fois sur les tests et sur le programme principal.
Nous n'avons pas observé de problèmes majeurs de mémoire par rapport à notre code, même si nous avons des erreurs que l'on suppose venir des bibliothèques. L'erreur la plus flagrante est celle des tests puisque nous contrôlons les dépendances et que nous n'utilisons pas OpenGL et GLFW sur les tests unitaires. Les erreurs du programme principal sont du même type que le listing 2 et sont disponibles dans le dossier /tests_result.

Il y a eu cependant des retours du client pour signaler des problèmes au niveau mémoire qui selon lui entraîne des exceptions de type "SegFault". Bien que nos tests ont normalement écarté ces possibilités, nous avons demandé aux clients plus d'informations sur ce problème particulier.

Listing 2: Extrait de la commande valgrind

```
valgrind -leak-check=full -show-leak-kinds=all
-log-file="../tests_result/valgrind_tests.txt" ./unit_tests
```

```
==17093== 32 bytes in 1 blocks are still reachable in loss record 1 of
    1
==17093==    at 0x4C31B25: calloc (in /usr/lib/valgrind/
    vgpreload_memcheck-amd64-linux.so)
==17093==    by 0x68377E4: _dlerror_run (dlerror.c:140)
==17093==    by 0x6837165: dlsym (dlsym.c:70)
==17093==    by 0x628943D: ??? (in /usr/lib/x86_64-linux-gnu/
    libGLdispatch.so.0.0.0)
==17093==    by 0x6286B68: ??? (in /usr/lib/x86_64-linux-gnu/
    libGLdispatch.so.0.0.0)
==17093==    by 0x4010732: call_init (dl-init.c:72)
==17093==    by 0x4010732: _dl_init (dl-init.c:119)
==17093==    by 0x40010C9: ??? (in /lib/x86_64-linux-gnu/ld-2.27.so)
```

7 Analyse du fonctionnement

La solution proposée permet de créer une planète entièrement procédurale, avec un rendu intéressant en low-poly pour une utilisation ludique, ce qui correspond à l'objectif de ce projet.

On dispose d'une architecture maintenable où il est notamment possible de changer les dépendances, comme par exemple le système de visualisation OpenGL par un de ses concurrents, DirectX ou Vulkan, ou encore la bibliothèque Libnoise par une autre plus performante avec un impact minimum sur le code.

Les éditeurs sont aussi un système qui permettra plus tard d'implémenter des générations de planètes plus complexe. L'une difficulté a été de trouver un moyen de faire en sorte que cela reste maintenable et de faciliter les futurs implémentions de l'éditeur. La logique de construction variant beaucoup d'une solution à une autre d'après nos recherches, nous avons fait le choix de simplement fournir des fonctionnalités de base afin qu'un futur développeur puisse implémenter son propre éditeur avec ses paramètres et ses règles.

Mais l'architecture du système de configuration devrait être améliorée. Même s'il a de nombreuses possibilités à l'avenir, il manque de souplesse et de maintenabilité. On peut changer l'implémentation pour passer de XML à JSON, mais il faudrait refaire tout le système de lecture puisque l'on utilise la bibliothèque pugiXML. Dû à l'actuelle implémentation, en cas de nouveau paramètre ou éditeur, il faut rajouter les cas un par un de lecture dans le XMLGenerator. Un système de tableau de paramètres et de types prédéfinis automatisé, où l'on ne s'occuperait plus d'implémenter la mécanique de lecture aurait été préférable. Cependant, cela aurait demandé beaucoup trop de temps pour être implémenté et nous sommes restés sur un simple système de lecture de XML, cette solution était suffisante.

Les résultats des tests ont aussi permis d'assurer une bonne base du logiciel, mais aussi et surtout de mesurer précisément l'impact au niveau des performances du logiciel. Certains sont encore à peaufiner pour tester certains cas inattendus comme lors de la gestion des fichiers de configuration, ou encore donner de mauvais paramètres aux éditeurs, mais on s'est assuré au minimum que ce soit fonctionnel en lisant la documentation.

Au niveau de l'esthétique, nous avons pu donner une grande liberté à l'utilisateur, ce qui est un grand point positif. Cependant, des points restent à améliorer, notamment sur la gestion des couleurs sur OpenGL qui fait que des faces de même couleurs se retrouvent séparées au niveau de la définition des bords de zones. Le niveau de l'océan aussi n'a pas été formellement défini et paramétrable, ce qui fait que certaines parties de la planète paraissent submergées. Aussi, pour faciliter les discussions avec le client, nous aurions dû lui proposer des références pour bien fixer le style graphique qu'il souhaitait.

8 Conclusion et perspectives

Nous avons réussi, par l'intermédiaire de ce projet à mettre en place une génération entièrement procédurale d'une planète. Cette génération produit un rendu intéressant en low-poly pour une utilisation ludique, ce qui correspond à l'objectif de ce projet. Néanmoins, plusieurs pistes d'améliorations sont envisageables :

Il serait intéressant d'implémenter des éditeurs générant des plaques tectoniques, rivières, biomes etc. Nous avons déjà réfléchi à des possibilités d'implémentation dans le fichier "docs/requirements/BiomesTheorie.pdf".

Comme l'a indiqué le client, il serait préférable que la gestion par cmake des dépendances trouve les bibliothèques déjà installées sur la machine plutôt qu'aller les chercher dans le répertoire "ext" (ou d'inviter à les télécharger si elles ne sont pas présentes), ce qui allégerait notre git. Nous avions procédé ainsi pour ne pas avoir à nous inquiéter des dépendances, peu importe la machine où l'on travaillait.

Le système de sauvegarde/chargement d'un fichier n'a pas toutes les fonctionnalités de départ que l'on aurait pu espérer. Actuellement, la sauvegarde est fonctionnelle (sans les couleurs dans le cas du format .obj) mais sans l'eau transparente puisqu'elle est implémenté directement dans le rendu. Le chargement d'un fichier n'a pas été implémenté mais son importance est amoindrie du fait de la possibilité de pouvoir re-créer une planète presque à l'identique en appelant le logiciel avec les mêmes paramètres de génération.

Aussi, une meilleur gestion de la caméra avec notamment une vue en première personne lorsque l'on se rapproche de la surface de la planète fait partie des fonctionnalités manquantes.

Enfin, le client a évoqué la possibilité de pouvoir appliquer dynamiquement des transformations sur la planète (élévation de terrain ou ajout de rivières par des interactions clavier/souris par exemple). Cela nécessiterait entre autres d'ajouter des méthodes dans l'interface Shape permettant de parcourir la connexité des sommets, mais aussi de revoir comment gérer cela au niveau de l'interface. Des solutions existent, mais elles nécessitent l'utilisation pour être efficace soit de structures d'accélération comme les kd-tree au niveau du model, soit des lectures de données renvoyées par la carte graphique.

9 Annexes

References

[1] Song Ho Ahn. **OpenGL Sphere**. http://www.songho.ca/opengl/gl_sphere.html, 2018. Accessed January 29th 2020.

[2] Pierre Bénard. **University Course Master 1 Informatic Image and Sound**. https://www.labri.fr/perso/pbenard/teaching/mondes3d/, Sept 2019. Accessed March 21st 2020.

[3] Joachim Schiele Keith Davies Owen Jacobson Cedric Pinson, Jason Bevins. **LibNoise, C++ OpenSource Noise Library**. http://http://libnoise.sourceforge.net/, 2007. Accessed March 31st 2020.

[4] Joey de Vries. **Learn OpenGL Camera**. https://learnopengl.com/Getting-started/Camera, Jun 2014. (Accessed january 2020).

[5] GNU Fdl. **Evenly distributed points on sphere**. http://web.archive.org/web/20120421191837/http://www.cgafaq.info/wiki/Evenly_distributed_points_on_sphere, 2011. Accessed January 29th 2020.

[6] Red Blob Games. **Delaunay + Voronoi on a sphere**. https://www.redblobgames.com/x/1842-delaunay-voronoi-sphere/, 19 Oct 2018. Accessed January 29th 2020.

[7] John Tsiombikas John F. Fay and Diederick C. Niehorster. **Freeglut : the free OpenGL Utility Toolkit**. http://freeglut.sourceforge.net/, Last update : 29 September 2019. (Accessed January 29th 2020).

[8] Acessed Marsh 2020 List of authors there https://github.com/glfw/glfw. **GLFW (GL Frame Work)**. https://www.glfw.org/, Last update : 20 january 2020. (Accessed January 29th 2020).

[9] Jordan Peck. **FastNoise, C++ OpenSource Noise Library**. https://github.com/Auburns/FastNoise, 2017. Accessed March 31st 2020.

[10] ScratchaPixel. **The Phong Model, Introduction to the Concepts of Shader, Reflection Models and BRDF**. https://www.scratchapixel.com/lessons/3d-basic-rendering/phong-shader-BRDF, 2017. Accessed April 3rd 2020.

[11] Patricio Gonzalez Vivo and Jen Lowe. **The Book of Shaders**. https://thebookofshaders.com/, 2015. (Accessed 30 january 2020).

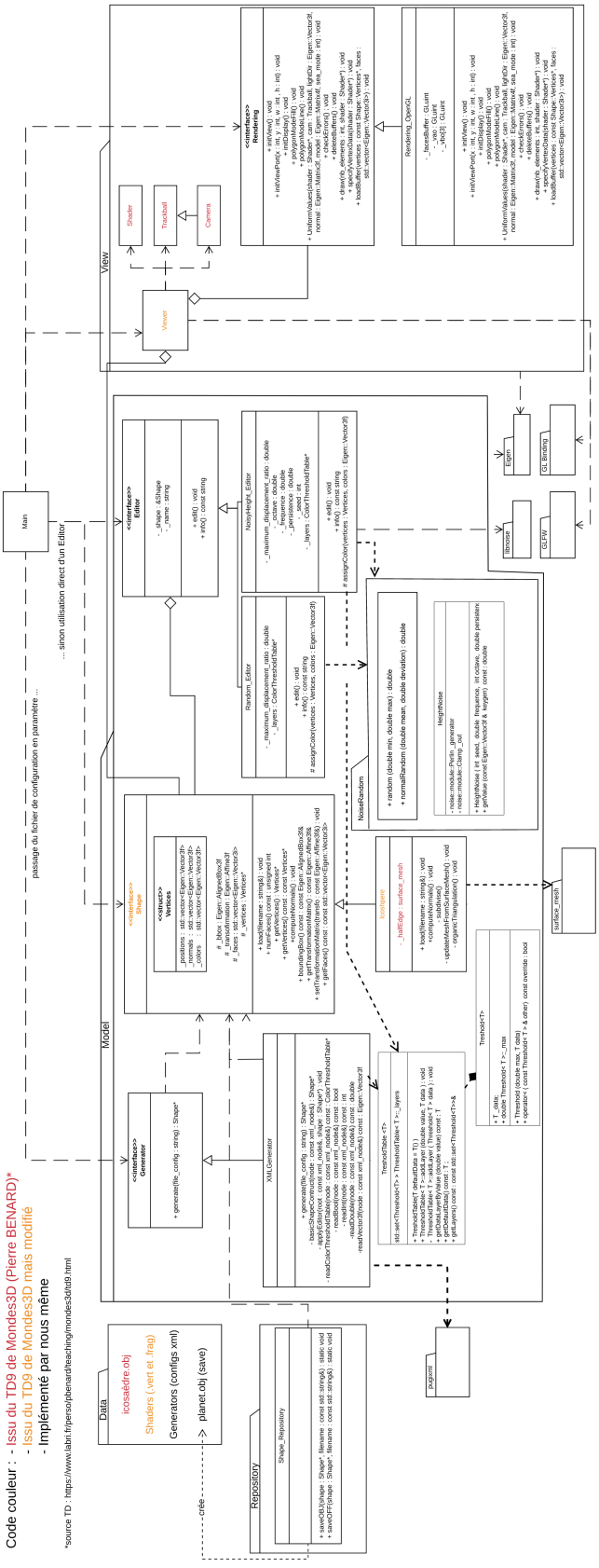

Figure 28: Architecture complète