Master 1 Informatique

Année 2019-2020

Projet de programmation

Groupe 21.1

Génération procédurale de planètes sphériques

Mémoire

Auteurs :
W.Daigremont
T.Guesdon
R.Montferme
L.Souvay
T.Videau

Table des matières

1 Contexte du projet

Ce document décrit le logiciel "Planet Generator" développé dans le cadre du Projet de Programmation 2019-2020. Il porte sur le sujet "Génération procédurale de planète sphérique" proposé par David Renault.

Le logiciel propose la génération et l'affichage d'une planète générée procéduralement, c'est à dire par un algorithme ayant des composantes aléatoires. Le rendu de la planète est un rendu appelé "low-poly" 5.3. Les planètes générées possèdent plusieurs plaques tectoniques, plusieurs environnements, des océans et du relief. Ces différentes caractéristiques ne varient pas après avoir été générées, il faut considérer l'affichage comme une capture à un instant t d'une planète, il n'y a donc pas de mouvement de plaques ou de montée des océans.

Sont également implémentés plusieurs contrôles utilisateurs permettant une manipulation de la planète générée, notamment une rotation autour de la planète et un zoom qui sont décrit plus en détail dans la partie 2.1.2. Il est possible de sauvegarder et recharger une planète précédemment générée.

Le développement de ce logiciel a uniquement un but d'éxperimentation, il pourra être réutilisés pour d'autres projets, comme par exemple dans le domaine du jeu vidéo ou de la création vidéo. Il pourrait également être utilisé par des artistes afin d'imaginer des mondes similaires à la planète Terre.

1.1 Analyse de l'existant

On traitera dans un premier temps des exemples d'implémentations qui répondent à une partie de nos besoins et les stratégies qu'ils utilisent. Dans un second temps, on décrira de manière plus détaillée les méthodes qui y ont été utilisées et qui pourraient être utiles à ce projet.

1.1.1 Exemples d'implémentations

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/*patchmatch.pdf***1.1.1.1** *Generationprocduraledepla*

Sebastian Lague a mis en ligne une série de vidéos didactiques expliquant comment générer procéduralement des planètes en utilisant le moteur de jeu Unity [1]. Au bout de 7 épisodes d'environ 15 minutes, S. Lague parvient à obtenir un résultat capable de satisfaire en partie les besoins de ce projet. On peut voir une capture d'écran figure 1 présentant le résultat. Les fichiers de son projet sont téléchargeables sur un dépôt git [1], avec la possibilité de les télécharger à l'état d'avancement de chaque épisode.

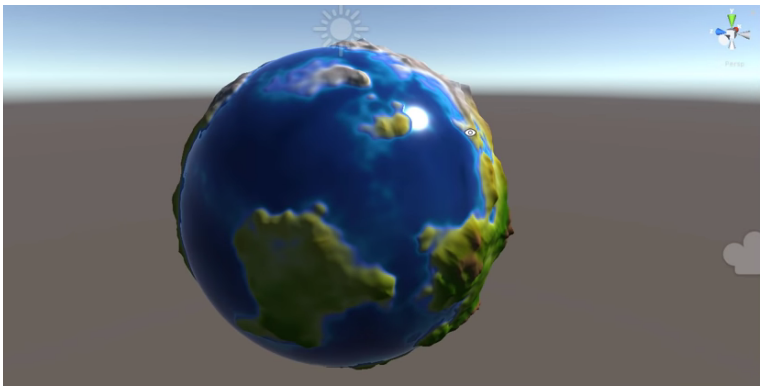

FIGURE 1 – Capture d'écran extraite de l'épisode final de la série de S. Lague [1]

S. Lague a utilisé plusieurs méthodes pour parvenir à ce résultat qui peuvent servir de base de solution à ce projet. Parmi celle-ci, on peut notamment relever la déformation des faces d'un cube pour obtenir une sphère. De cette manière, il peut découper les faces du cube à sa guise avant de les déformer. On peut également noter l'utilisation d'un filtre bruit cohérent pour définir l'altitude de sa planète, dans son cas le filtre "simplex noise", qui peut être également remplacé par un bruit de Perlin. Voir [2] .

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/*patchmatch.pdf***1.1.1.2** *Legnrateurdeplantejavascr*

Découpage de la planète en faces

1. https ://github.com/SebLague/Procedural-Planets

On cherche ici à séparer la planète, qui est une sphère, en une multitude de faces, afin de leur attribuer à chacune une couleur, et si possible une altitude. Andy Gainey a fait face à la même problématique, et il décrit sa solution sur son site [3], avec laquelle il a réussi à atteindre le nombre de 36 002 faces, tout en permettant des contrôles de caméra 5.4 fluide au sein d'un navigateur web.

Découpage de la planète en plaques tectoniques

Dans un premier temps, Adam Gainey a défini les plaques tectoniques avec un nombre aléatoire de faces. Il a ensuite rajouté à chacune des plaques tectoniques des paramètres procéduraux qui permettent de définir leurs mouvements. Il a assigné à chaque plaque si elle était océanique ou continentale de manière aléatoire là aussi, tout comme l'élévation de ces plaques (élévation négative pour une plaque océanique et élévation positive pour une plaque continentale).

Génération de l'altitude

Pour générer de l'altitude, Gainey a procédé en deux étapes. Il a d'abord calculé l'élévation aux frontières des plaques tectoniques, par rapport aux mouvements des plaques tectoniques voisines(séparation ou collision) ainsi que leur type (océanique ou continentale).

En suite, pour ce qui est des points à l'intérieur des plaques, il a utilisé plusieurs paramètres pour calculer l'élévation :

- La distance entre les points et la frontière la plus proche
- la distance entre les points et le centre de la plaque
- Répartition des biomes en fonction du climat.

Voir la référence [3] pour des explications plus détaillées.

Génération et répartition des biomes

voir 5.1 Après avoir calculé l'altitude, la température ainsi que le niveau de précipitation, Andy Gainey déclare avoir assez d'éléments pour pouvoir créer des biomes :

- Une région chaude avec une altitude faible et peu de précipitations devient un désert. Si la région avait des précipitations élevées, elle serait devenue une forêt tropicale
- Une région avec peu d'élévation, une température et des précipitations modérées devient une prairie
- Les régions froides avec peu d'humidité deviennent des toundra
- Les régions à haute altitude deviennent des montagnes rocheuses ou neigeuses selon la température et les précipitations.

1.1.2 Algorithmes existants

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/patchmatch.pdf**1.1.2.1** $Affichageetrendud'unesph$

Il existe de très nombreuses manières d'afficher une sphère en 3D, ne sera détaillé ici que les possibilités qui sont envisageables dans le cadre de ce projet.

Moteur de jeu

Il existe plusieurs moteurs de jeu permettant l'affichage d'une sphère en 3D, on peut, par exemple, citer Unity qui a été utilisé par Sebastian Lague (voir 1). Ces moteurs de jeu nous permettraient de passer très vite à la génération procédurale sans nous être beaucoup impliqués dans l'affichage et le rendu de la sphère. Or nous y impliquer est un des buts majeurs de ce projet.

Utilisation d'une bibliothèque de fonctions

Une autre possibilité est d'utiliser une bibliothèque de fonctions qui permettrait d'éviter d'avoir à coder les fonctions basiques de calcul d'images 3D. On peut par exemple citer OpenGL, DirectX ou Vulkan.

"From Scratch"

Une dernière possibilité serait de partir de zéro, sans bibliothèque ou moteur externe. Cependant, suivre cette méthode prendrait beaucoup plus de temps, et il serait peu probable que cela soit à portée de l'équipe responsable de ce projet dans le temps imparti.

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/patchmatch.pdf**1.1.2.2** $Lesmthodesdepavaged'unes$

Le pavage ou "tesselation" en anglais est définit comme tel dans la version en ligne de Merriam-Webster 2020 [4] :

"a covering of an infinite geometric plane without gaps or overlaps by congruent plane figures of one type or a few types"

"Un recouvrement d'un plan géométrique infini sans trous ou superpositions avec des figures planes d'un ou plusieurs types"

La méthode basée sur l'icosaèdre

Max Tegmark détaille dans son article "An icosahedron-based method for pixelizing the celestial sphere" [5] l'utilisation d'un icosaèdre pour paver une sphere. L'icosaèdre étant le polyèdre régulier possèdant le plus de faces, c'est à dire 20 (Icosaèdre Régulier [2]). Un polyèdre régulier étant lui même un polyhèdre dont toutes les faces sont des polynomes réguliers et dont tous les sommets sont de

2. polyèdre régulier à 20 faces triangulaires

même degrés.
Max Tegmark explique donc qu'il est possible d'utiliser un polyèdre régulier englobant une sphere pour la paver. Il faut alors paver les faces du polyèdre puis projeter ces faces sur la sphere. On peut par exemple utiliser un cube, mais le résultat serait moins précis qu'avec un icosaèdre, voir figure 2 pour en avoir un aperçu. Les points représentent les intersections présentes sur le polyèdre de départ.

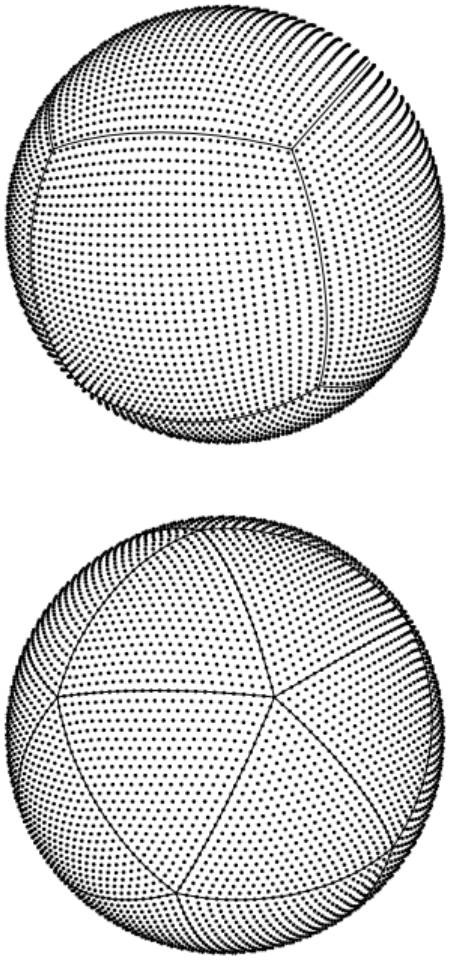

FIGURE 2 – Méthode basée sur le cube (en haut) et l'icosaèdre (en bas) de pavage d'une sphere

Les diagrammes de Voronoï

Un diagramme de Voronoï est également une forme de pavage d'un plan qui se construit à partir d'un ensemble de points. On peut par exemple utiliser des points obtenus avec la méthode de l'icosaèdre évoquée plus tôt, ou utiliser un ensemble de points aléatoires pour obtenir un rendu plus chaotique.

Une fois l'ensemble P des points définits, on construit une "cellule" pour chaque point. Une cellule d'un point p de l'ensemble P est composée de tous les points du plan ayant pour plus proche voisin dans P le point p. Voir figure 3.

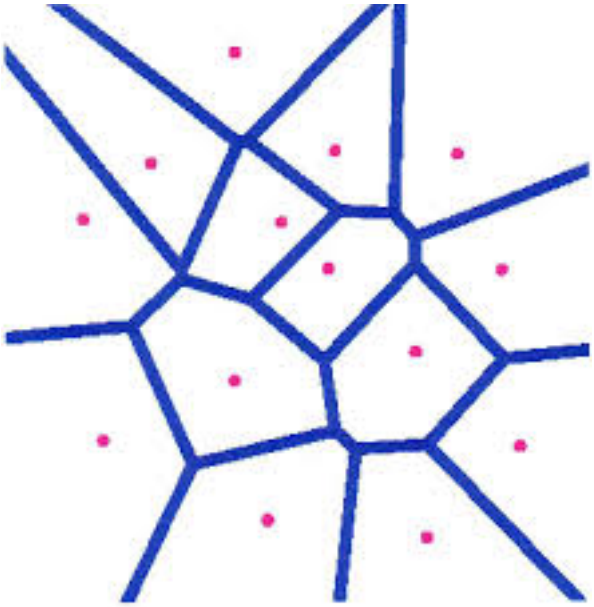

FIGURE 3 – Exemple de pavage crée à l'aide d'un diagramme de Voronoï [6]

Points aléatoires sur une sphère

On peut définir un point sur une sphère à l'aide de deux valeurs, $\lambda \in [-180, 180]$ et $\phi \in [-90, 90]$. Intuitivement, on pense à choisir aléatoirement ces deux valeurs, cependant on obtient alors une répartition non uniforme, avec une concentration des points plus importante autour des pôles de la sphère [7].

Répartition uniforme

Il existe néanmoins un algorithme permettant d'obtenir une répartition uniforme de ces points. Celui-ci est l'algorithme du "meilleur candidat" de Mitchell [8], cet algorithme prend en entrée un nombre de points à placer sur un plan. Pour chaque point, il génère un échantillon de points aléatoires, puis sélectionne dans cet échantillon le point le plus éloigné de tous les points déjà placés. On obtient de cette manière une répartition uniforme des points sur la surface de la sphère.

Bruit de Perlin [2]

La fonction bruit de Perlin est une fonction en n dimensions qui génère un bruit pseudo-aléatoire en fonction du temps. Celle-ci permet de représenter de manière plus réaliste certains évènements naturels tels que des chaînes de montagnes ou de la matière organique. On trouve plusieurs exemples d'implémentations de cette méthode en ligne. On pourra utiliser par exemple l'implémentation java développée par Ken Perlin en 2002, qu'il met à disposition sur sa page de l'université de New York. [9].

Le bruit de Perlin renvoie des valeurs comprises dans l'intervalle $[0, 1]$, on peut alors changer d'intervalle pour que les valeurs soient comprises dans une intervalle $[min, max]$ avec min et max choisis.

Il existe également l'algorithme "simplex noise" qui est une amélioration du bruit de Perlin qui est plus efficace lorsque l'on a un nombre élevé n de dimensions.

1.2 Besoins fonctionnels

1.2.1 Rendu

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/patchmatch.pdf **1.2.1.1** *Moteurderendu*

Nous utiliserons OpenGL (voir 1.3 pour les détails de version) afin de réaliser le rendu de notre planète. OpenGL nous permet de placer des formes géométriques dans une scène 3D grâce à un panel de fonctions, et se charge de calculer l'image qui apparait à l'écran. Nous pouvons placer également des sources de lumières et positionner le point depuis lequel on regarde la scène, appelé "caméra" (voir5.4). Plusieurs membres de l'équipe ont déjà manipulé cet outil, et celui-ci a été recommandé par le client. De plus il s'agit d'une bibliothèque très bien documentée, c'est pour ces raisons qu'il a été choisi de l'utiliser. Il existe également plusieurs exemples disponibles en ligne de planètes générées procéduralement et affichées via OpenGL. On trouve par exemple une implémentation utilisant C++ et OpenGL affichant une planète procéduralement générée en temps réel [10]. De plus cette planète n'est pas affichée avec un rendu low-poly mais avec un nombre de polygones que l'on peut déduire bien supérieur aux 10.000 qu'il est souhaité atteindre, en ajoutant à cela le fait que plusieurs planètes soient affichées cela fourni une preuve quant à la réalisation de ce projet avec OpenGL.

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/patchmatch.pdf **1.2.1.2** *Renduvisuel*

Le visuel de la planète sera en low poly, c'est à dire qu'elle sera modélisée avec une quantité assez réduite de polygones colorés,(c.f.4). Ce rendu ne représente pas l'aspect génération (hauteur, plaque tectonique, rivière) mais représente l'aspect graphique. Les couleurs seront attribuées en fonction de la hauteur et du biome. La planète contiendra aussi des rivières et des nuages afin d'obtenir un meilleur aspect visuel. Pour obtenir ce rendu, des shaders [3] seront utilisés. Cela permettra d'avoir de la réflexion lumineuse dans les couleurs ou permettre de visualiser des reflets pour l'eau ou de la transparence pour les nuages. Il permet aussi de jouer sur plusieurs paramètre comme la lumière ou l'ombre.

3. un *shader* ou nuanceur est un programme informatique permettant de paramétrer les processus de synthèse graphique d'image

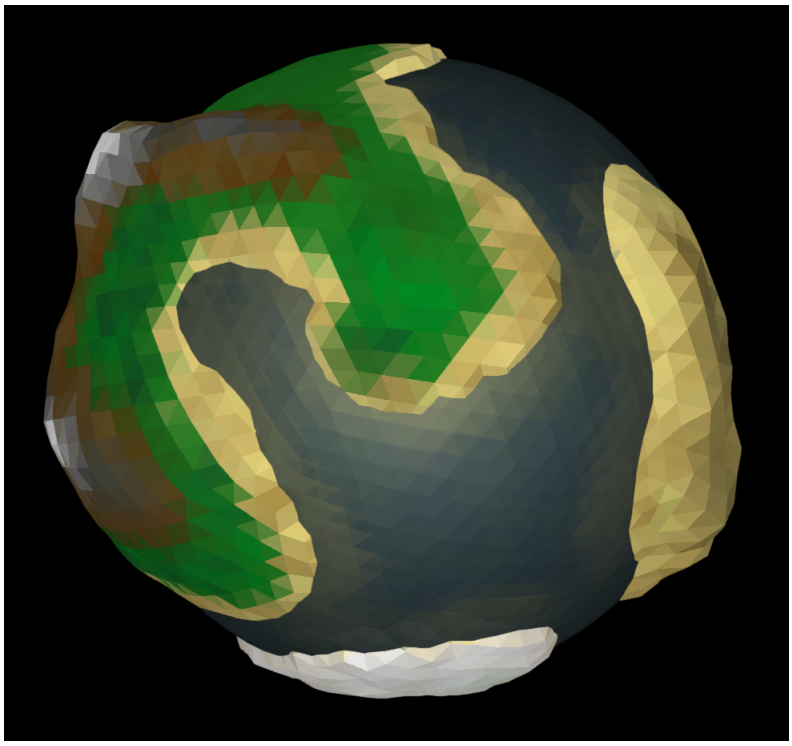

FIGURE 4 – Exemple d'un rendu visuel réalisé grace au logiciel PlanetPainter

1.2.2 Génération procédurale de la planète

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$*AR*/*patchmatch.pdf***1.2.2.1** $Dcoupagedelaplanteenfaces$

Pour dessiner la planète, il a été décidé de s'inspirer des travaux de Song Ho Ahn [11] qui décrit plusieurs méthodes pour afficher une sphère avec OpenGL et C++. Celui-ci décrit notamment la méthode basée sur l'icosaèdre qui est la plus intéressante, en effet pour pouvoir diviser notre planète en 10.000 faces que l'on pourra ensuite colorer comme souhaité pour représenter la planète. De plus les faces obtenues sont triangulaires, ce qui permet de mieux représenter les côtes que la méthode basée sur le cube par exemple.
Afin d'obtenir le rendu low poly demandé par le client,on cherche à diviser la planète en un nombre n de faces, avec pour objectif d'atteindre 10.000 faces. Dans la partie 1.2.1 on a pu établir qu'avec OpenGL et en suivant les travaux de Song Ho Ahn il était possible de créer une sphère avec une multitude de triangles. On utilisera la couleur de ces faces pour représenter les différents environnements, forêts, océans, déserts, etc ... Ces faces découpées seront les plus petites subdivisions de la planètes, elles seront incluses elle même dans d'autre subdivisions de la planète (plaque tectoniques, biomes). Ces faces découpées seront les plus petites subdivisions de la planètes, elles seront incluses elle même

dans d'autre subdivisions de la planète (plaque tectoniques, biomes).

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/patchmatch.pdf **1.2.2.2** *Dcoupageenplaquetectoniqu*

Dans la génération de planète, une plaque tectonique sera modélisée par une composante connexe constituée de faces. C'est à dire un regroupement de faces ayant au moins une arête en commun. On observe sur la planète terre une quinzaine de plaques principales (voir 5). Sera donc créé un nombre de plaque tectoniques variant légèrement autour de ce nombre.

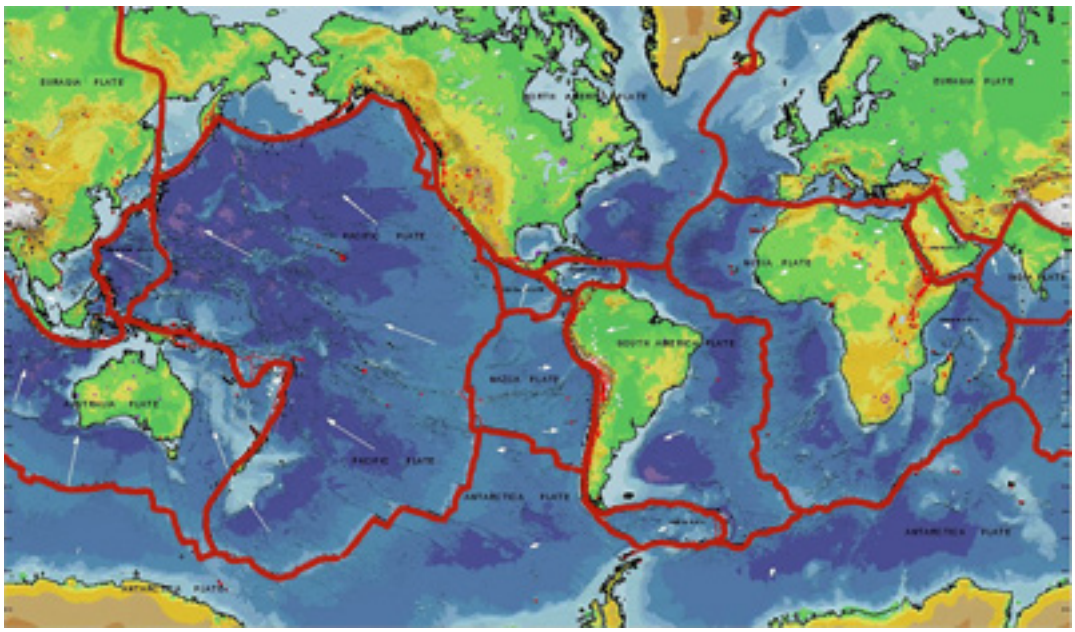

FIGURE 5 – Principales plaques tectoniques terrestres [12]

Pour créer aléatoirement ces plaques tectoniques un algorithme a été imaginé. Il prend en entrée le nombre de plaques que souhaité et l'ensemble des faces de notre planète.

Algorithm 1 Creer_Plaques(entier nb_plaques, liste<Face> faces)

```
liste < liste < Face >> plaques
for i : 0 → nb_plaques do
  plaques[i].push(faces.pop(random))
end for
i ← 0
while !faces.est_vide() do
  plaques[i].push(faces.pop(premiere_case_adjacente(faces, plaques[i])))
  if i < nb_plaques then
    i++
  else if i == nb_plaques then
    i ← 0
  end if
end while
retourner plaques
```

Avec la fonction $premiere\_case\_adjacente(faces, plaques[i])$ renvoyant la première case $c_1 \in faces$ étant adjacente à une case $c_2 \in plaques[i]$. En utilisant cette fonction on évite de devoir choisir une face aléatoirement puis de vérifier qu'elle soit adjacente à notre plaque, ce qui nous permet d'être sûr que notre algorithme se termine.

On souhaite faire une modélisation très simplifiée des plaques tectoniques réelles, c'est pour cela que sera donné à chaque plaque une direction et une vitesse, bien que celles-ci ne se déplaceront pas réellement. Ces deux valeurs permettront de simuler le rapprochement ou l'éloignement des plaques sans les déplacer réellement. On considérera que lorsque deux plaques se rapprochent l'une de l'autre, elles formeront une chaîne de montagne à leur frontière.

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/patchmatch.pdf**1.2.2.3** *Gnrationdel'altitude*

A été évoqué la possibilité d'utiliser le bruit de Perlin afin de générer du relief, cependant on souhaite lier la génération de montagne aux plaques tectoniques. C'est à dire que l'on souhaite faire apparaître des montagnes là où deux plaques tectoniques entrent en collision. Pour cela deux solutions sont envisagées.

La première, plus simple, consiste à ajouter un point p a une certaine distance alt dans la direction du vecteur normal [4] des faces frontalières des plaques. Cette distance alt correspondra à l'altitude des chaînes de montagnes, que varieront en fonction de la vitesse de collision des plaques. On construira ensuite les montagnes en créant des triangles ayant pour sommet le point p et les sommets de chaque arrête de la face. Sera ajouté ensuite une altitude légèrement inférieure à alt aux faces voisines, puis aux voisines de celles-ci, etc... De cette manière on

4. vecteur orthogonal par rapport à un plan

obtiendra une altitude supérieure à 0 sur un nombre x de cases en fonction de la vitesse de décroissance que l'on appliquera. Les faces ayant gardé une altitude supérieure à 0 après la génération de l'altitude seront considérée comme des faces océaniques, auxquelles on attribuera une couleur bleue sans tenir compte des modifications pouvant être liés aux biomes.

La deuxième, plus compliquée, implémentable si il y a suffisament d'avance sur les autres items, consiste à appliquer un bruit de Perlin sur l'ensemble de la planète pour définir l'altitude des points p, puis à appliquer un produit de convolution d'une fonction f sur cette fonction bruit de perlin. La fonction f prendrait la forme d'une cloche au niveau des chaînes de montagnes.

On choisit la fonction bruit de perlin et non le "simplex noise" puisque l'on travaille avec une fonction en 3 dimensions, les optimisations de "simplex noise" ne seront pas utiles (voir 1.1.2.2) et la fonction bruit de perlin est mieux documentée, notamment au niveau des exemples d'implémentations.

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$*AR*/*patchmatch.pdf***1.2.2.4** *Gnrationderivire*

On cherchera à générer des rivières afin d'apporter un meilleur visuel. Pour cela, on prendra des points aléatoires dans des biomes relativement humide forêt tropicale, prairie, taiga, puis un chemin de point sera calculé afin de permette d'arriver au niveau de la mer.

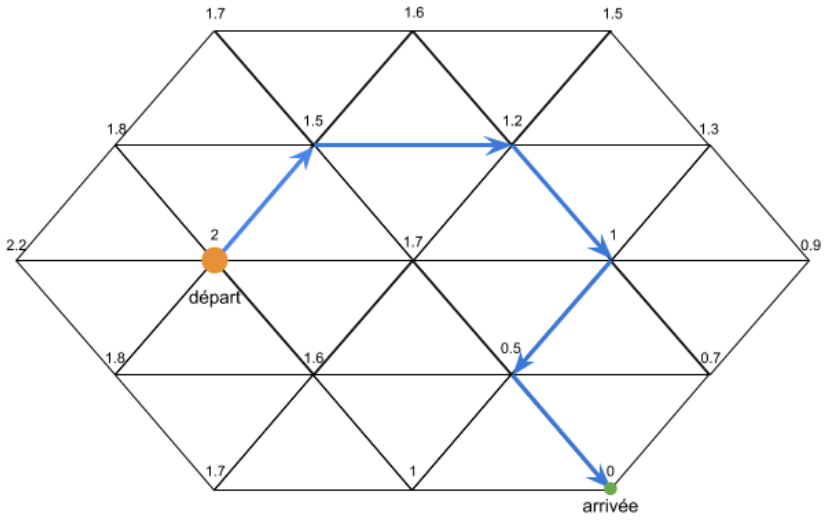

FIGURE 6 – Modèle d'une rivière sur le maillage de la sphère

1.2.3 Interactions utilisateurs

Plusieurs interactions utilisateurs seront implémentées, celles-ci devront être intégralement utilisable avec un clavier d'ordinateur portable classique. Cer-

taines pourront être également utilisées à l'aide de la souris.

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/patchmatch.pdf**1.2.3.1** $Zoometdzoomdelaplante$

Le logiciel doit proposer un zoom (rapprochement de la planète) afin de permettre une meilleure vision de la planète. Ce zoom devra être limité, afin de ne pas se rapprocher, ou s'éloigner trop de la planète. Dans le cas d'un rapprochement maximal, une rotation positive autour de l'axe X sera appliquée afin de proposée une vision plus ésthétique de la planète. Pour cela on définira une variable $zoom\_max$ légèrement supérieur au rayon r de la planète. Si un zoom est déclenché alors que la valeur de zoom est égale à $zoom\_max$ alors la rotation est appliquée.

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/patchmatch.pdf**1.2.3.2** $Rotationhorizontaleetverti$

Il sera également proposé la possibilité de tourner la caméra autour de la planète. La rotation pourra être appliquée sur l'axe Y ou X afin de permettre de visualiser la planète sous tous les angles. Nous utiliserons pour cela les méthodes $AngleAxis$ et $Translation$ d'OpenGL qui nous permettent de faire tourner un objet autour d'un point.

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/patchmatch.pdf**1.2.3.3** $Modefilsdefer$

Pour permettre au client de mieux visualiser les différentes faces qui composent la planète, il sera également implémenté un affichage en mode "fils de fer", qu'il sera possible d'activer via une touche clavier. Pour cela nous créerons un deuxième shader OpenGL n'affichant que les arêtes des faces.

1.2.4 Sauvegarde de la planète

Il sera également nécessaire de proposer un système de sauvegarde et restauration de la planète. Pour cela nous créerons un fichier texte contenant les différentes valeures utiles à la génération de la planète.

1.3 Besoins non fonctionnels

1.3.1 Etude des contraintes

En prenant en compte les contraintes matérielles et logicielles imposées par la machine du client, il semble tout à fait faisable de produire une application étant capable de générer une planète et de l'afficher en temps maitrisé. On ne vise pas les 30 images par secondes. Le code du projet doit être écrit dans le langage de programmation C++. Il a été vu précedement qu'il est possible de générer et afficher un planète comportant plus de 10000 points uniquement à l'aide de code javascript s'exécutant sur un navigateur. Il est donc pensable de pouvoir en faire de même à l'aide de la bibliothèque OpenGL qui est mieux optimisée qu'un navigateur web. [13] Le client a demandé que le logiciel puisse s'éxécuter sur sa machine, qui a pour OS Gentoo Linux. La dernière version d'OpenGL supportée par la machine du client est la version 4.4. On utilisera la version 3.3 d'OpenGl pour de raison de compatibilité avec le CREMI et les machines

personnelles de l'équipe. L'application doit pouvoir fonctionner avec moins de 16Gb de mémoire et 1Gb de mémoire vidéo dédiée sur la carte graphique.

1.4 Priorités

Les différentes tâches de ce projet sont répartie sur un échelle de priorité définit ainsi :

- Priorité **1** : Tâche à terminer impérativement.
- Priorité **2** : Tâche non essentielle mais à terminer de préférence.
- Priorité **3** : Tâche non prioritaire à terminer après les tâches de niveau 1 et 2

1.4.1 Priorité 1

- Affichage d'une sphère composée de 10.000 faces.
- Rotation de la sphère
- Déplacement de la sphère sur l'axe Z (zoom)
- Implémentation basique des plaques tectoniques (Ensemble de faces et direction).
- Implémentation basique des biomes (Ensemble de faces avec une couleur).
- Rendu visuel de la planète.
- Sauvegarde et chargement de la planète.

1.4.2 Priorité 2

- Ajout d'altitude à la frontière des plaques.
- Ajout du climat, chaud à l'équateur, froid aux pôles.
- Répartition des biomes en fonction du climat.

1.4.3 Priorité 3

- Ajout des nuages.
- Ajout des rivières.
- Ajout d'un panel de couleurs au lieu d'une unique couleur pour les biomes.
- Compatibilité de notre système de sauvegarde avec le second groupe.

2 Logiciel

2.1 Fonctionnalités

A l'éxecution, notre logiciel génère procéduralement une planète. Divers contrôle utilisateurs sont disponibles afin de visualiser cette planète. Un système de sauvegarde de la planète à aussi été mis en place.

2.1.1 Génération de la planète

Pour générer la planète, nous partons d'un icosaèdre à 20 faces triangulaires car nous allons plus facilement obtenir une sphère en subdivisant un icosaèdre 2. Nous subdivisons ensuite chaque face en 4 faces plus petites de même tailles de manière récursive. Après avoir suffisament subdivisé l'icosahèdre selon le paramètre planet_smooth présent dans le fichier properties.txt (voir 2.1.1.1) , il suffit de normaliser la distance entre chaque point et le centre de l'icosahèdre pour obtrenir un volume se rapprochant à une sphère. Plus l'on subdivise plus le volume se rapproche alors d'une sphère, voir 7,8 ci-dessous.

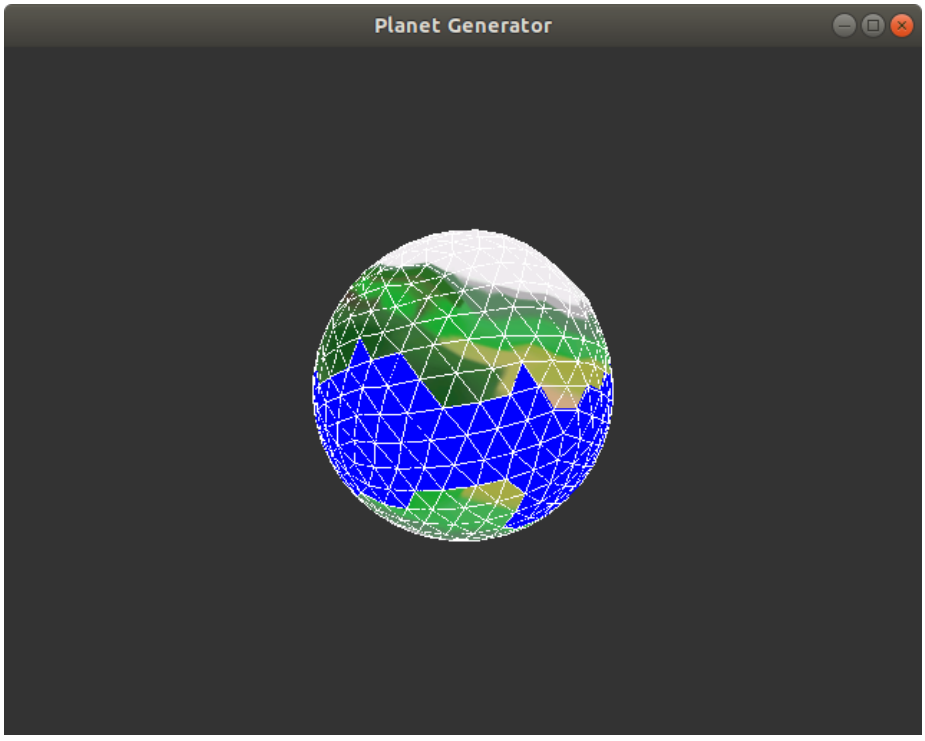

FIGURE 7 – Planète sphérique avec 3 subdivisions

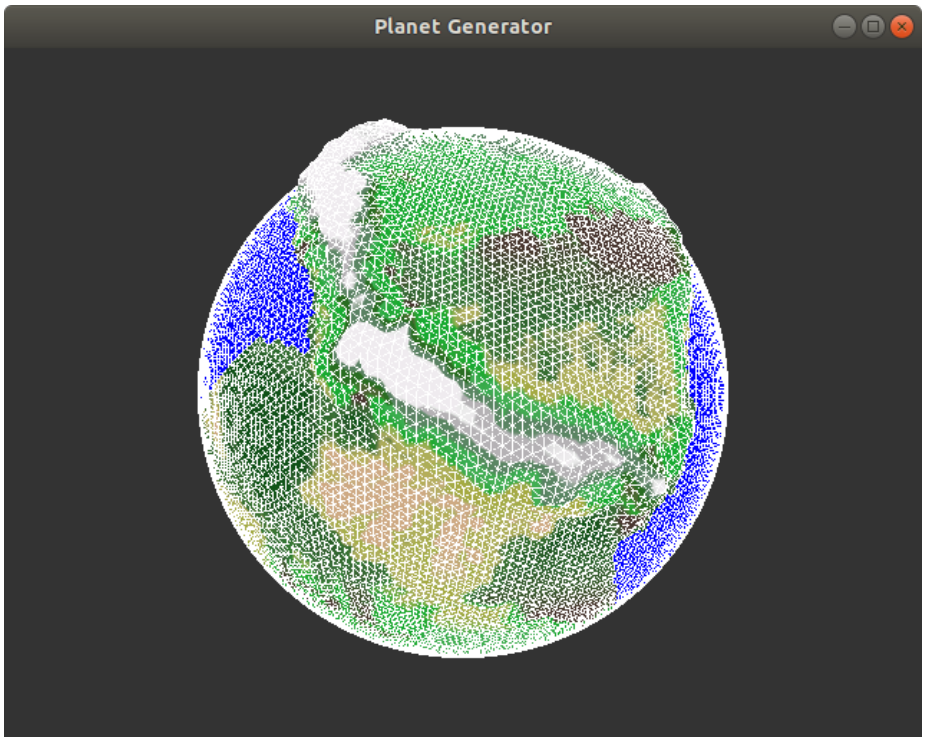

FIGURE 8 – Planète sphérique avec 6 subdivisions (zoomée)

Après avoir généré cette "sphère", on la divise en un nombre choisi de plaques tectoniques. Chaque plaque peut etre soit continentales, soit océanique. Lors de la génération des plaques, on atribue à chacune d'entre elles un "déplacement" selon le plan normal à la surface de la sphère. Le déplacement d'une plaque n'est qu'indicatif, la plaque ne bouge pas.

Une fois les plaques tectoniques générées, on attribue une humidité a chaque face appartenant aux plaques continentales.

Avec la bibliothèque SOIL, on viendra ensuite charger une texture et lire sur cette dernière la couleur d'une face en fonction de la température et de l'humidité lui ayant été attribuée. Cette étape a lieu lors du rendu de la planète à l'aide des shaders en OpenGL.

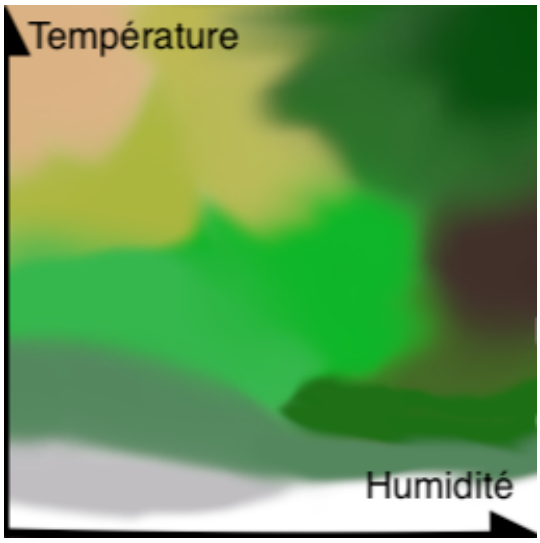

FIGURE 9 – Texture fonction de l'humidité et de la température

Enfin, on utilise les déplacements liés aux plaques pour pour générer les montagnes. Si deux plaques se "rapprochent", on applique alors une élévation a leur limites, ce qui crée des montagnes.

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$*AR/patchmatch.pdf***2.1.1.1** *Paramtrage*

Il est possible de paramétrer différents éléments lors de la génération de la planète à l'aide du fichier "properties.txt" qui se trouve dans le dossier "src/" du programme. Ce fichier permet notamment de choisir le nombre de subdivision de la sphère, la hauteur maximale des montagnes, le nombres de plaques, le nombre de zone d'humidité, la seed utilisée pour la génération (voir 2.1.2.8).

```
1   seed=-1
2   planet_radius=2
3   planet_smooth=6
4   plate_number=10
5   plate_continental=40
6   height_max=0.15
7   height_interval=0.02
8   humidity_zone=100
9   humidity_smooth=[5,15]
10  humidity_variation=[-3,3]
```

FIGURE 10 – Fichier "properties.txt" par défaut

2.1.2 Interactions utilisateurs

A l'aide du clavier et de la souris, l'utilisateur peut s'approcher/s'éloigner de la planète et la faire pivoter selon l'axe X et l'axe Y.

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/patchmatch.pdf**2.1.2.1** $Zoometdzoomdelaplante$

Il est possible de se rapprocher de la planète et de s'en éloigner en utilisant la molette de la souris, pour les utilisateurs n'en utilisant pas il est également possible d'utiliser les touches "page down" et "page up" du clavier. Lorsque l'on se rapproche suffisament de la planète l'angle de caméra change afin de permettre à l'utilisateur de mieux apprécier le relief.

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/patchmatch.pdf**2.1.2.2** $Rotationhorizontaleetverti$

Il est possible d'afficher la planète sous tous les angles en controllant sa rotation autour de l'axe X et l'axe Y. Les touches haut et bas permettent la rotation autour de l'axe X, les touches droite et gauche permettent la rotation autour de l'axe Y.

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/patchmatch.pdf**2.1.2.3** $Affichageterrestre$

L'affichage terrestre est l'affichage par défaut. La planète y est affichée comme on la verrait depuis l'espace. Pour choisir cet affichage il faut utiliser la touche &.

FIGURE 11 – Affichage terrestre

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/patchmatch.pdf**2.1.2.4** $Affichagedeszonesd'humid$

L'affichage des zones d'humidité est accessible via la touche é. Cet affichage permet de distinguer les différentes zones d'humidité, du plus humide (blanc), au moins humide(noir).

FIGURE 12 – Affichage des zones d'humidité

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$*AR*/*patchmatch.pdf***2.1.2.5** $Affichagedestemperatures$

L'affichage des temperatures est accessible via la touche ". Cet affichage permet de distinguer les variations de temperatures, du plus chaud(blanc) au plus froid(noir).

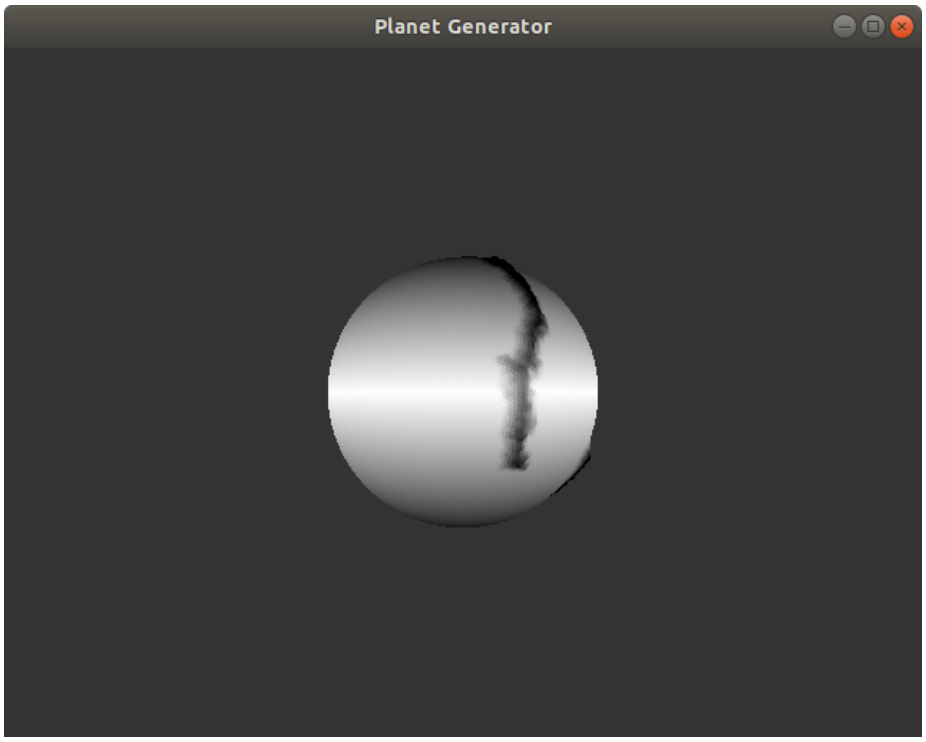

FIGURE 13 – Affichage des temperatures

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$*AR*/*patchmatch.pdf***2.1.2.6** *Affichage"fildefer"*

L'affichage en mode "fil de fer" est accessible via la touche '. Cet affichage met en valeur les arrêtes des faces, afin de mieux distinguer les faces de la planète.

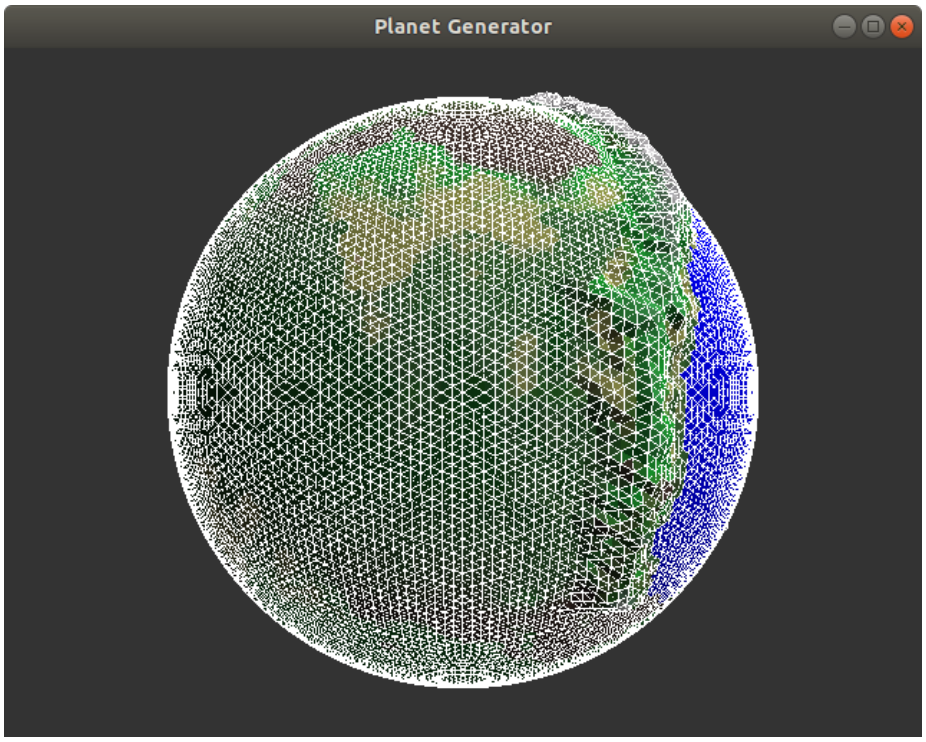

FIGURE 14 – Affichage en mode "fil de fer"

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$*AR*/*patchmatch.pdf***2.1.2.7** *Affichagedesplaquestecton*

L'affichage des plaques tectoniques est accessible via la touche (. Cet affichage met en valeur les frontières des plaques tectoniques afin de distinguer plus facilement ces mêmes plaques.

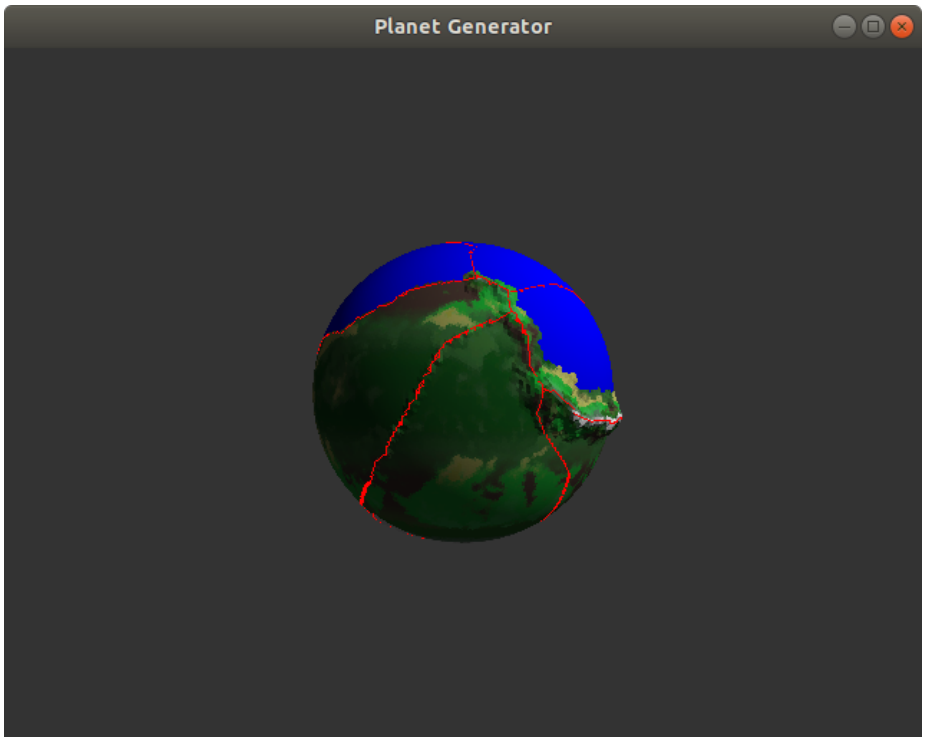

FIGURE 15 – Affichage des plaques tectoniques

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$*AR/patchmatch.pdf***2.1.2.8** *Sauvegarde*

Il est possible de sauvegarder la graine utilisée pour générer l'aléatoire de la planète en appuyant sur la touche 's'. Il est ensuite possible de charger la planète correspondante à la graine en entrant la graine sauvegardée dans le fichier de paramétrage dans le champ *seed* en y remplaçant la valeur -1 qui génère une graine aléatoire.

2.1.3 Scénario d'utilisation

L'utilisateur doit pouvoir lancer le programme depuis le terminal, puis lancer la génération d'une planète, ou charger une planète déjà existante. Une fois le chargement ou la génération terminée, l'utilisateur pourra alors visualiser la planète en tournant autour ou en se rapprochant d'elle à l'aide des touches du clavier. L'utilisateur doit aussi être en mesure de sauvegarder la planète qu'il visualise sous forme de fichier .txt. Avant de générer la planète, il sera demandé à l'utilisateur de choisir un nom à cette dernière, le fichier de sauvegarde portera alors ce même nom. La génération de la planete doit se faire en un temps inférieur à 30 secondes.

2.2 Architecture

2.2.1 Diagramme des paquets

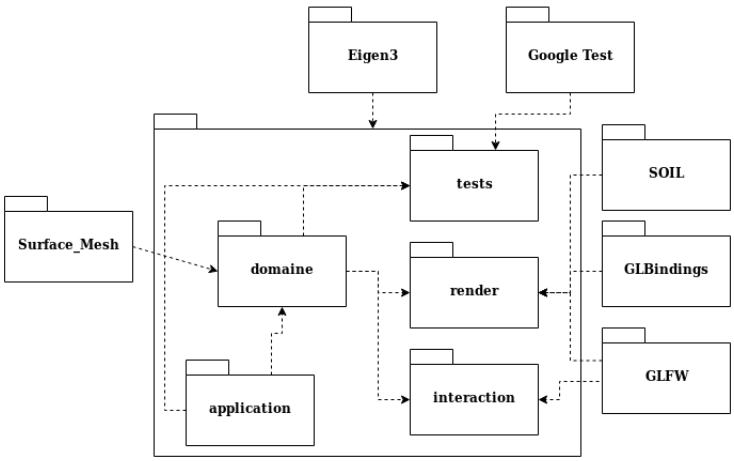

FIGURE 16 – Diagramme des paquets

L'application utilise plusieurs bibliothèques externes pour son fonctionnement :

- Nous utilisons la bibliothèque GLFW afin de faire les interactions clavier,souris et fenêtre.
- La bibliothèque GLBindings permet de faire le lien entre l'implémentation d'OpenGL fournie par le pilote de la carte graphique et le C++.
- La bibliothèque SOIL va nous permettre de charge et manipuler des textures.
- La bibliothèque Eigen3 nous permet de réaliser l'ensemble des calculs d'algèbre linéaire.
- La bibliothèque Surface_Mesh, permet elle la manipulation plus intelligente du maillage triangulaire.
- Enfin, la bibliothèque GoogleTest [14] sera notre bibliothèque pour pouvoir faire des tests.

Ces bibliothèques sont utilisées par les différents paquets de notre application ces derniers expliqués plus en détails par la suite.

2.2.2 Diagramme des classes

Nous avons essayé de découper le plus possbile notre code pour le rendre facilement maintenable et le plus compréhensible. Grace à ce découpage, il est simple de changer l'implémentation d'un paquet sans impacter l'implémentation des autres paquets.

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/*patchmatch.pdf***2.2.2.1** *Paquetdomaine*

Notre architecture a été pensée pour respecter au mieux le modèle de développement appelé DDD "Domain Directed Development", propre à la programmation orientée objet. Ainsi, le paquet Domaine peut etre compilé seul car il ne dépend pas des autres paquets. Le paquet Domaine représente le coeur de l'application. On y trouve la centrale classe *Planet* qui contient les liens vers les différentes factories qui généreront les caractéristiques de cette planète. La fonction *generatePlanet* génère la sphère et execute les fonctions de génération des factories.

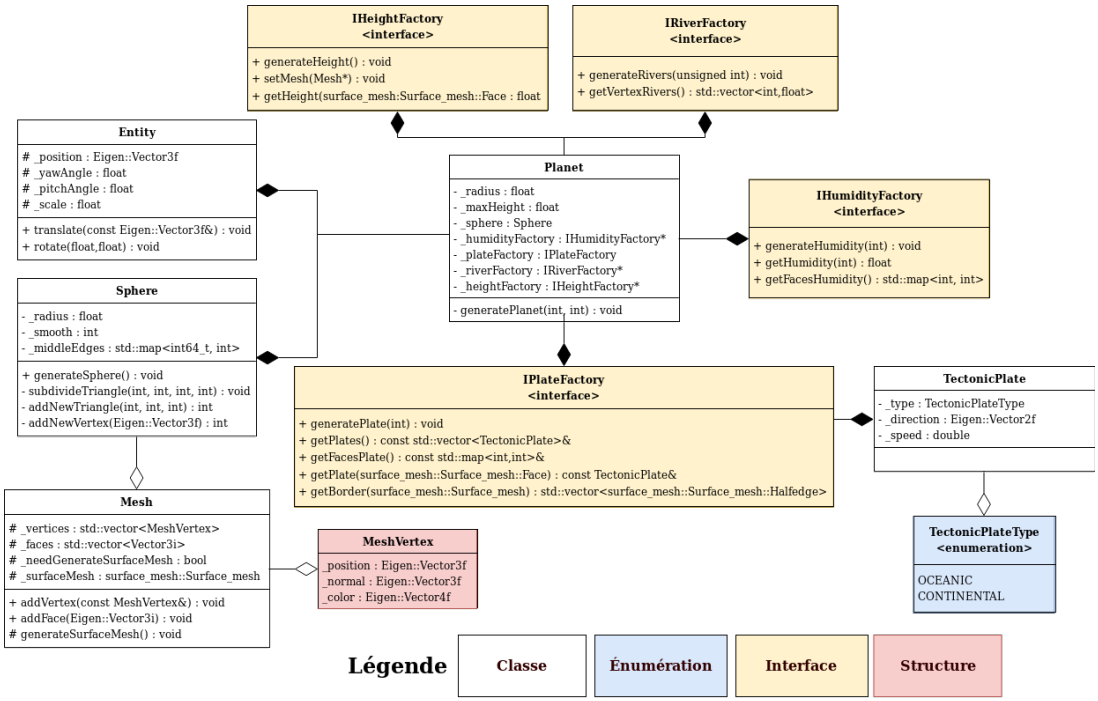

FIGURE 17 – Package Domaine

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/*patchmatch.pdf***2.2.2.2** *PaquetApplication*

Le paquet Application contient les classes qui implémentent les différentes interfaces du paquet domaine. C'est ici que l'on trouve les algorithmes de génération procédurale de la planète. Les différentes classes peuvent être remplacées aisément dans le cas où une meilleure implémentation serait développée. Ce paquet dépend du paquet Domaine.

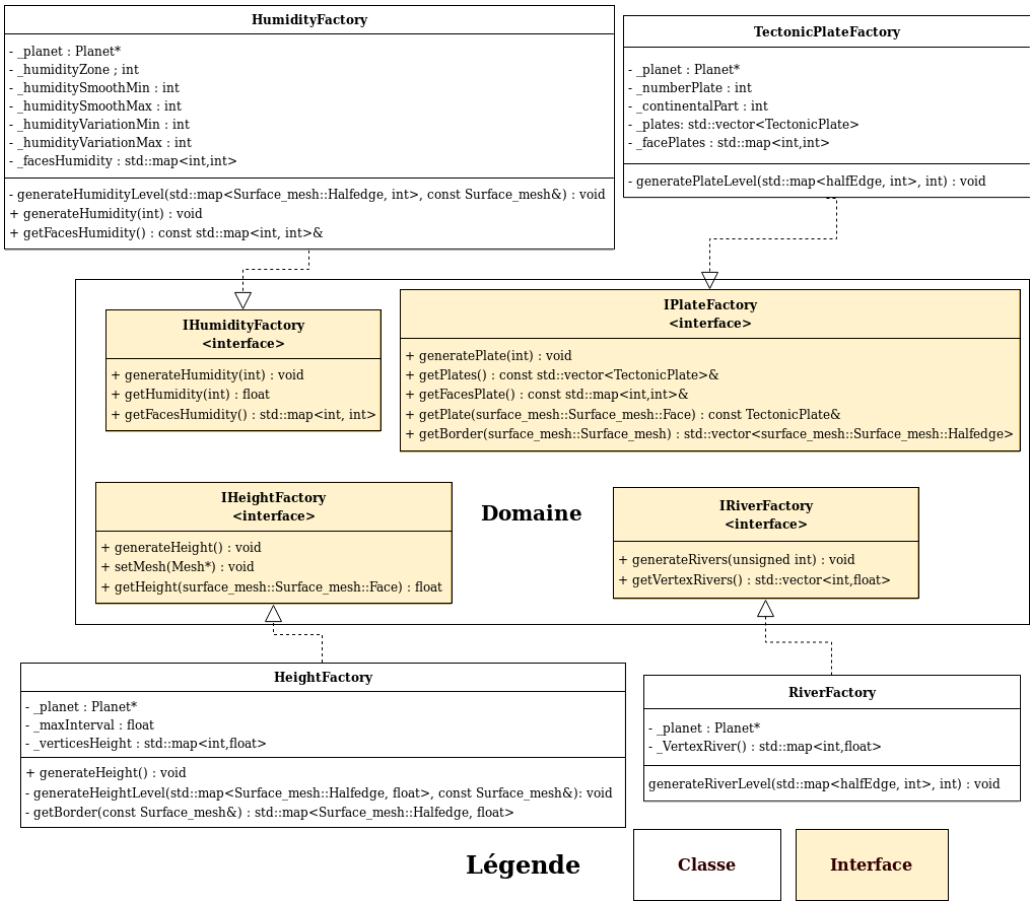

FIGURE 18 – Package Application

Nous avons aussi implémentés dans le paquet un fichier qui va nous permettre de sauvegarder et charger la planète (Figure 19).

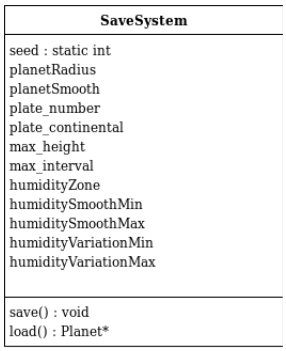

FIGURE 19 – SaveSystem

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/*patchmatch.pdf* **2.2.2.3** *PaquetRender*

Le paquet render contient les classes qui utilisent OpenGL pour afficher la planète. Il dépend du paquet domaine duquel il récupère les données de la planète. Le constructeur de la classe Renderer contient les fonctions de chargement des textures et de positionnement de la lumière. La fonction dessinant le mesh de la planète est la fonction $drawMesh()$, qui est appelée par la fonction $drawScene()$ qui positionne également la caméra.
Nous n'avons pas implémenté la classe Shader, nous l'avons prise de l'archive du TD de Mondes3D de l'Université de Bordeaux [15]. Nous avons aussi pris la classe Caméra de cette archive, mais nous avons dû implémenté les fonctions $lookAt()$ et $rotateAroundTarget()$. Nous avons implémenté Renderer et Viewer mais nous nous sommes encore une fois inspiré de l'archive.

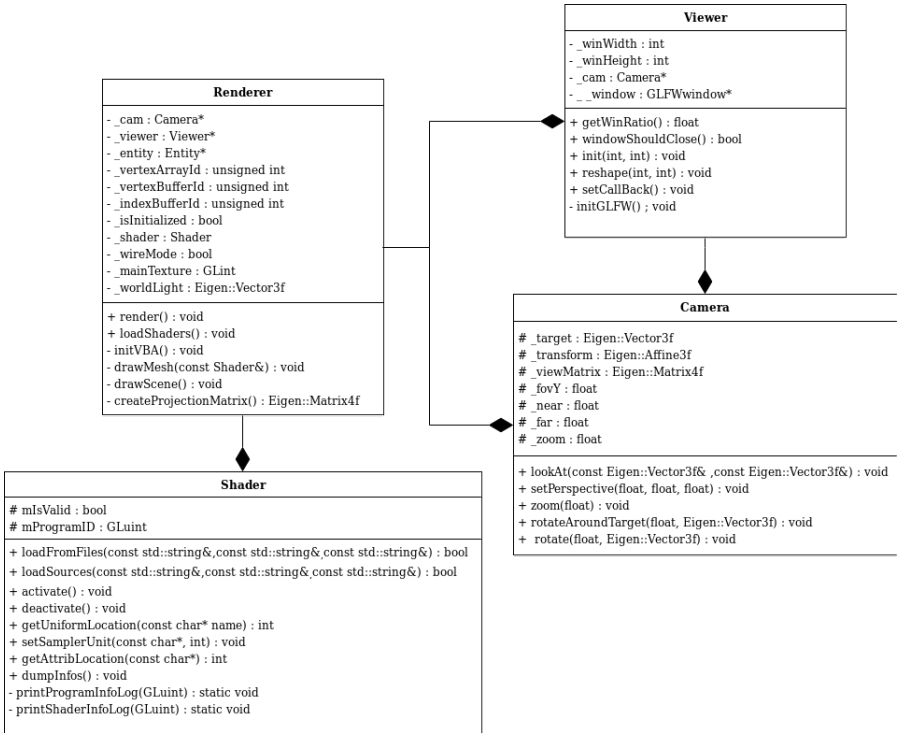

FIGURE 20 – Paquet Render

https://gfx.cs.princeton.edu/pubs/Barnes$_2$009$_P$AR/*patchmatch.pdf* **2.2.2.4** *PaquetInteraction*

La paquet interaction contient le code permettant de faire le lien entre une entrée utilisateur, utilisant le clavier ou la souris, et une fonction du Renderer, voir 2.2.2.3. Il dépend donc du paquet render. La classe *trackball* permet de gèrer le glissement de la souris. La pression d'une touche de clavier est gérée par la fonction *keyPressed* de la classe Interaction.
Ici aussi, l'archive de Mondes3D [15] nous a servi. Nous n'avons pas implémenté Trackball ni l'énumération TrackMode. On a créé la classe Callback mais avons pas implémenté les fonctions de la classe.

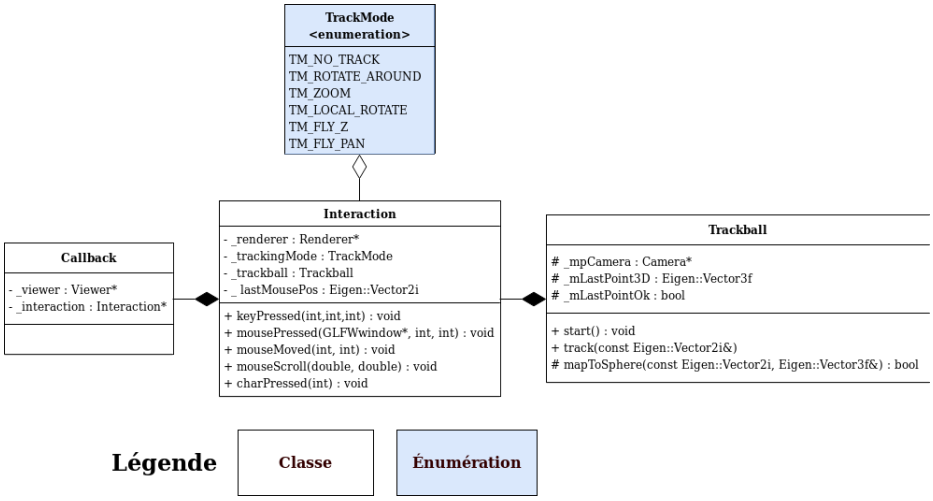

FIGURE 21 – Paquet Interaction

2.2.3 Diagramme de séquence

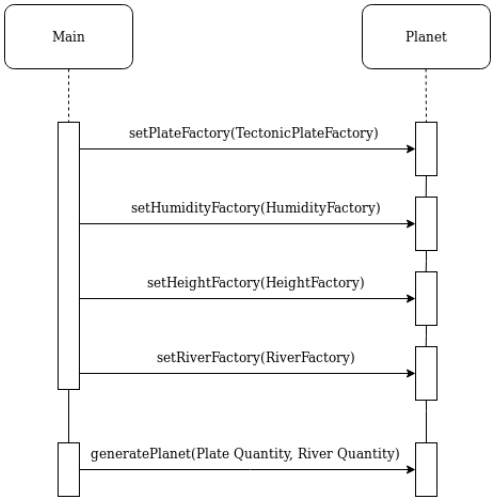

FIGURE 22 – Génération de la planète

Toute la génération de la Planète s'effectue comme présenté sur la figure **??**. On commence par fournir à la planète des *Factories*, chacune chargée de la génération d'un aspect de la planète, puis on appelle la fonction lançant la génération de la planète et qui va faire usage de chaque *Factory* donnée.

2.3 Points techniques d'implémentation

2.3.1 Création des plaques tectoniques

Pour créer les plaques tectoniques, on utilise la structure Halfedge, présent dans la bibliothéque Surface Mesh. Halfedge va nous permettre de récupérer les voisins des faces de manière efficace.
Une halfedge h stocke :

- Le sommet vers lequel elle pointe (h->vtx)
- La face auquel elle appartient (h->face)
- Son halfedge opposé (h->opposite)
- Le prochain halfedge (h->next)

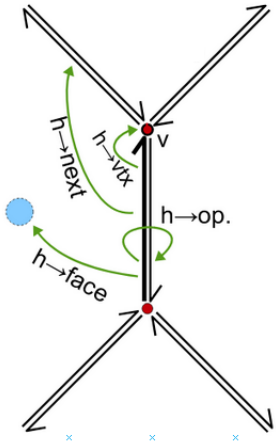

FIGURE 23 – Halfedge [16]

Pour pouvoir récupérer tous les voisins d'une face, on parcourt toutes les halfedges d'une face à l'aide de h->next, puis on prend l'halfedge opposé de l'halfedge sur lequel on est pour avoir la face voisine. La complexité pour trouver les voisins d'une face est de O(3) car nos faces sont des triangles, ce qui est rapide.
On calcule le nombre de faces pour une plaque de manière aléatoire, et tant que le nombre de faces n'est pas atteint, on continue à chercher les voisins.
On utilise la même technique pour pouvoir créer les zones d'humidités.

2.3.2 Génération du relief

Pour créer les reliefs, nous appliquons un bruit aléatoire comrpis entre un minimum et un maximum pouvant etre modifié par l'utilisateur. Ce bruit est augmenté en bordures de plaques pour générer des chaines de montagnes. Plus précisément, le calcul du relief en bordure de plaques dépend du mouvement des deux plaques voisines. Si les deux plaques vont l'une vers l'autre, une chaine de montages est formée.

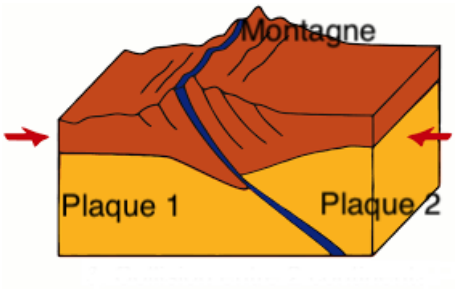

FIGURE 24 – Génération des montagnes

Nous ne génèrons de reliefs que si les plaques se rapprochent sinon, les points gardent leur relief de base.

2.3.3 Rappels sur OpenGL

Notre application utilise OpenGL, une interface de programmation afin d'exploiter les performances et capacités des cartes graphiques. Pour cela il faut donc que nous utilisions des *shaders*[5]. Ces *shaders* sont des programmes qui s'exécuteront sur le processeur graphique d'une carte graphique.

Les *shaders* reçoivent en entrée des données envoyés depuis le processeur. Ces données sont chargées en mémoire vidéo. C'est ce qui se passe notamment dans notre programme dans la méthode *initVBA()* de la classe *Renderer*.
On utilise souvent 2 types de shaders : *fragment shader* et *vertex shader*, mais dans notre cas on en utilise un troisième comme évoqué dans la partie 2.3.4. En OpenGL les images sont dites **rastérisé**, c'est à dire que l'on projette un objet sur un tableau de pixel représentant l'image résultante. Le passage d'un objet 3d à une image s'appelle alors la **rastérisation**.

Le *vertex shader* est le premier type de shader qui s'exécute ici. Il s'exécute pour chaque point du maillage. Il reçoit des données en entrée depuis la mémoire vidéo. Dans notre application, c'est du côté CPU via la méthode *drawMesh()* que l'on défini l'emplacement des données dans la mémoire vidéo et quand doivent s'exécuter les shaders. Il envoie ensuite différents type de données en sortie au

5. Traduisible par Nuanceurs

prochain type de shader.

Le *fragment shader* est généralement le dernier type de shader exécuté. Ce dernier travail sur chaque pixel de l'image. Les données qu'il reçoit sont souvent obtenues après l'étape de rastérisation. Ce shader travaille sur chaque pixel et c'est souvent ce dernier qui permet de colorer les objets 3d rasterisés.

Comme évoqué plus tôt, dans notre application, nous utilisons aussi un 3ème type de shader qui est décrit 2.3.4. On peut voir sur la figure 25 un schéma de notre pipeline OpenGL :

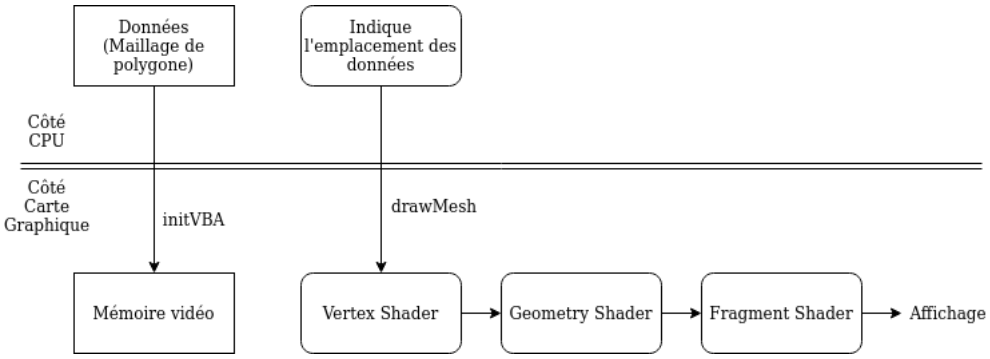

FIGURE 25 – Pipeline graphique de l'application

2.3.4 Utilisation du geometry shader

Dans notre programme, nous utilisons le geometry shader. Ce dernier fonctionne entre le vertex shader et le fragment shader. Et a la particularité de créer de nouvelles géométries à la volée en utilisant la sortie du vertex shader en entrée. Il nous permet principalement d'avoir un rendu low poly propre sans avoir à multiplier les sommets. En effet, pour obtenir des ombres et des couleurs low poly, comme on peut le voir sur le schema 26, il faut que les normales de chaque triangle du modèle soient séparées. Pour cela, il y a deux méthodes, la première consiste à séparer tous les triangles du mesh comme on peut le voir sur le schema 27, cela implique une multiplication des vertices par 6. La deuxième méthode consiste à réaliser une moyenne des trois normales de chaque triangle dans le geometry shader.

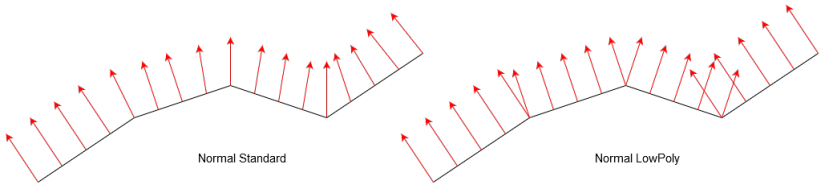

FIGURE 26 – Modèle des normals

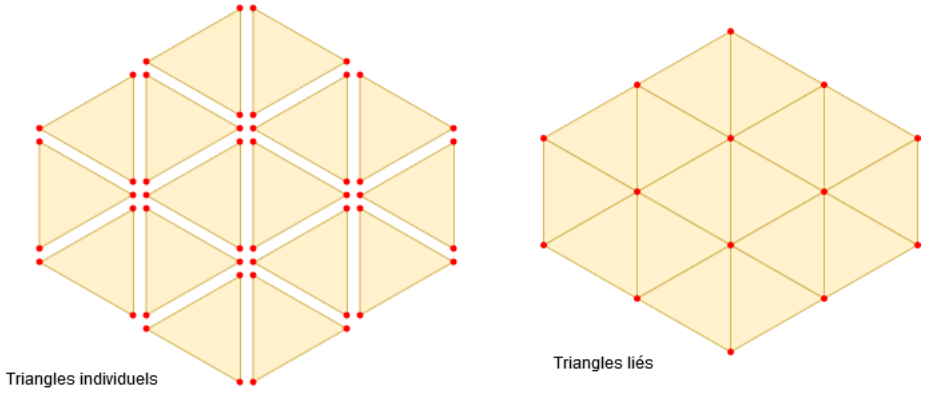

FIGURE 27 – Modèle des triangles

Nous avons choisi la deuxième méthode car nous ne pouvons pas séparer les triangles pour trouver les arrêtes voisines du mesh lors de la génération. Cela implique en contrepartie, qu'il faut une machine avec une carte graphique compatible. C'est pour cela que nous avons deux versions de shader, une avec le rendu low poly et le geometry shader, et une avec un rendu standard sans geometry shader. Nous savons que la machine du client est compatible mais nous avons mis en place ce système car certaines personnes du groupe n'avaient pas de machine compatible. Il nous permet aussi d'avoir l'océan seulement là ou se trouve les plaques océaniques, où le relief est à 0 sur les trois sommets du triangle.

3 Analyse du fonctionnement et tests

3.1 Analyse du fonctionnement

3.1.1 Génération de la planète

Nous avons analysé le comportement de l'application, plus précisément, le temps nécéssaire pour la génération d'une planète. Dans un premier temps nous nous sommes concentré sur l'importance du "détail" de la planète correspondant au nombre de subdivison de la sphère. Des mesures de temps nous ont permis de tracer cette courbe :

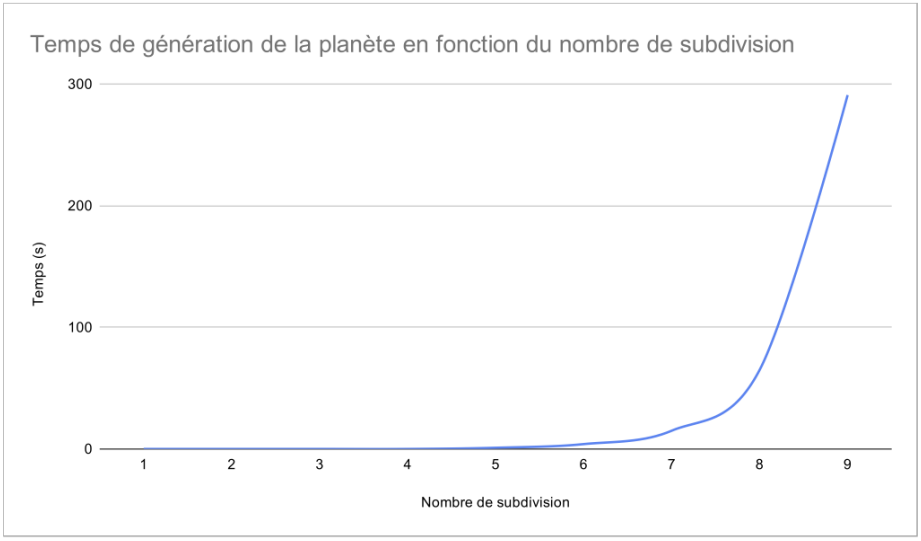

FIGURE 28 – courbe de performance 1

Puis nous nous sommes penchés sur l'importance du nombre de plaques et de zones d'humidité. Nous avons fait varier ces deux composantes de manière similaire à la variation du nombre de faces de la sphère afin de tester le comportement du programme dans des conditions difficiles. Pour d'obtenir des données exploitables, les mesures ont été faites sur des planètes ne comportant pas d'océans.

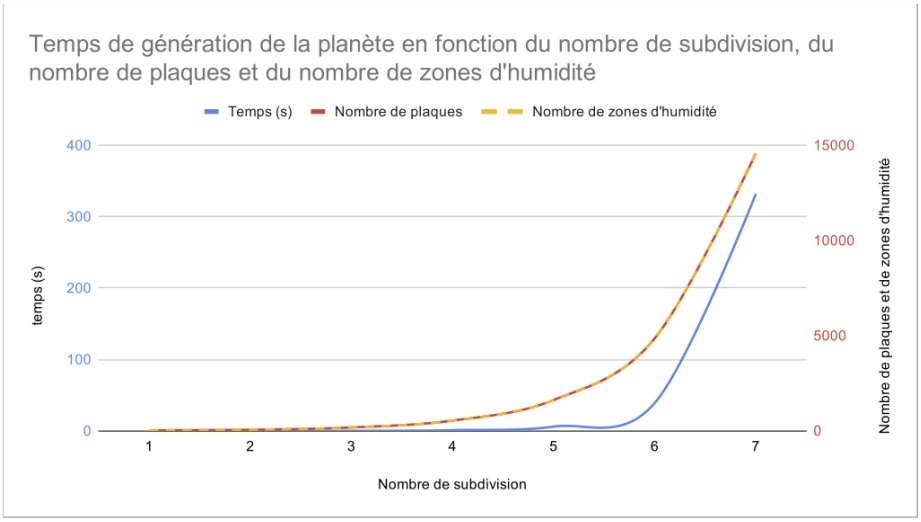

FIGURE 29 – courbe de performance 2

Nous pouvons observer deux courbes de performances de croissances exponentielles. Cela est du au fait que la subdivision récursive produit un nombre exponentiel de faces. Aussi, la deuxiéme coubre a un comportement similaire à la première, elle commence simplement à monter plus tot. Nous pouvons donc en déduire que le nombre de plaques et de zones d'humidité influent sur le temps de génération. Enfin, nous n'avons pas pu prendre d'avantage de mesures car le temps de génération etait trop long (plus de 15minutes), le programme peut alors etre stoppé par le système d'exploitation, c'est le cas sur MacOs par exemple.

Avec des paramètres résonable permetant d'obtenir une planète correctement détaillée :
- nombre de subdivison = 6
- nombre de plaques = 50
- nombre de zones d'humidité = 100
- pourcentage de plaques continentales = 40

Le temps de génaration d'une planète n'excède pas 7s, ce qui est bien en dessous des 30 secondes que nous nou étions fixés.

3.1.2 Fuites Mémoire

Afin d'analyser les fuites mémoires de notre programme, nous avons utilisé l'outil Valgrind et plus précisément, son outil Memcheck. La figure 30 présente un partie de la sortie de la commande : valgrind -leak-check=full

```
==24985== HEAP SUMMARY:
==24985==     in use at exit: 245,601 bytes in 2,807 blocks
==24985==   total heap usage: 435,398 allocs, 432,591 frees, 106,245,226 bytes allocated
==24985==
==24985== 112 (56 direct, 56 indirect) bytes in 1 blocks are definitely lost in loss record 1,654 of 1,704
==24985==    at 0x4837B65: calloc (vg_replace_malloc.c:752)
==24985==    by 0x94B5CCB: ???
==24985==    by 0x94B5C0B: ???
==24985==    by 0x94B2E05: ???
==24985==    by 0x94B2ECC: ???
==24985==    by 0x94B39B3: ???
==24985==    by 0x94B0313: ???
==24985==    by 0x94B2BB3: ???
==24985==    by 0x94ADC38: ???
==24985==    by 0x94ADC07: ???
==24985==    by 0x9C33ECD: ???
==24985==    by 0x9C331F6: ???
==24985==
==24985== 408 bytes in 1 blocks are definitely lost in loss record 1,669 of 1,704
==24985==    at 0x4837B65: calloc (vg_replace_malloc.c:752)
==24985==    by 0x4FD70FB: _XimOpenIM (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==24985==    by 0x4FD6D45: _XimRegisterIMInstantiateCallback (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==24985==    by 0x4FBE35A: XRegisterIMInstantiateCallback (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==24985==    by 0x7D8E13: _glfwPlatformInit (in /home/lsouvay/Documents/pdp-20-pg-sirius/src/build-v2-Desktop-Du00e9faut/pdp-plangen-20-g1)
==24985==    by 0x7CDF9D: glfwInit (in /home/lsouvay/Documents/pdp-20-pg-sirius/src/build-v2-Desktop-Du00e9faut/pdp-plangen-20-g1)
==24985==    by 0x7C9408: Viewer::initGLFW() (in /home/lsouvay/Documents/pdp-20-pg-sirius/src/build-v2-Desktop-Du00e9faut/pdp-plangen-20-g1)
==24985==    by 0x7C92FE: Viewer::init(int, int) (in /home/lsouvay/Documents/pdp-20-pg-sirius/src/build-v2-Desktop-Du00e9faut/pdp-plangen-20-g1)
==24985==    by 0x7CC06E: main (in /home/lsouvay/Documents/pdp-20-pg-sirius/src/build-v2-Desktop-Du00e9faut/pdp-plangen-20-g1)
==24985==
==24985== 1,688 (136 direct, 1,552 indirect) bytes in 1 blocks are definitely lost in loss record 1,684 of 1,704
==24985==    at 0x4837D7B: realloc (vg_replace_malloc.c:826)
==24985==    by 0x4FC37A9: ??? (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==24985==    by 0x4FC3C04: ??? (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==24985==    by 0x4FC552E: ??? (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==24985==    by 0x4FC5D47: _XlcCreateLC (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==24985==    by 0x4FE66DF: _XlcUtf8Loader (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==24985==    by 0x4FCD1FD: _XOpenLC (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==24985==    by 0x4FCD42A: _XrmInitParseInfo (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==24985==    by 0x4FB451F: ??? (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==24985==    by 0x4FB887D: XrmGetStringDatabase (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==24985==    by 0x7D8EAB: getSystemContentScale (in /home/lsouvay/Documents/pdp-20-pg-sirius/src/build-v2-Desktop-Du00e9faut/pdp-plangen-20-g1)
==24985==    by 0x7D8D6B: _glfwPlatformInit (in /home/lsouvay/Documents/pdp-20-pg-sirius/src/build-v2-Desktop-Du00e9faut/pdp-plangen-20-g1)
==24985==
==24985== LEAK SUMMARY:
==24985==    definitely lost: 600 bytes in 3 blocks
==24985==    indirectly lost: 1,608 bytes in 35 blocks
==24985==      possibly lost: 0 bytes in 0 blocks
==24985==    still reachable: 243,393 bytes in 2,769 blocks
==24985==         suppressed: 0 bytes in 0 blocks
==24985== Reachable blocks (those to which a pointer was found) are not shown.
==24985== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==24985==
```

FIGURE 30 – Valgrind –leak-check=full

On peut voir ici les fuites mémoires pour lesquelles les bits sont définitivement perdus ou indirectement perdus.

La figure 31 présente cette fois-ci une partie de la sortie de la commande valgrind –leak-check==full –show-leak-kinds=all et nous permet de voir les bits qui sont toujours accessibles.

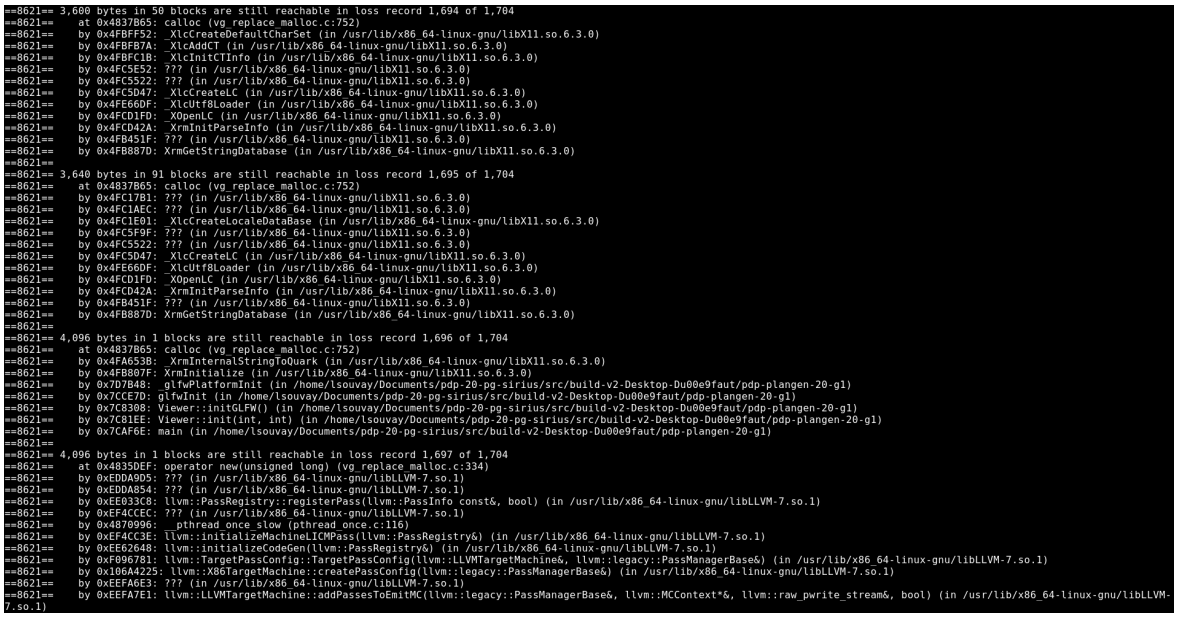

FIGURE 31 – Valgrind –leak-check=full –show-leak-kinds=all

Nous n'avons pas pu régler ces fuites mémoires et elles n'ont pas l'air d'être solubles car elles semblent liées à la bibliothèque GFLW.

3.2 Tests

Les fichiers contenant nos tests se trouvent dans le dossier src/v2/tests. Nos tests utilisent Google Test [14].

3.2.1 Test des paramètres

Ce test vérifie que tous les paramètres fournit en entrée dans properties.txt excepté la seed ont bien été récupérées dans les variables correspondates.
Pour ce faire, on charge la planète décrite par les paramètres de properties.txt (load()). Puis, on récupère les valeurs des paramètres de properties.txt, afin de voir si elles correspondent à celles qui ont été chargées par la planète juste avant. Si la valeur récupéré avec la fonction load() n'est pas la même que celle présente dans le fichier, alors le test renvoie faux, sinon il renvoie vrai.
Dans notre cas, tous les tests sur les paramètres sont passés, c'est-à-dire que chaque paramètre de properties.txt a bien été récupéré par la fonction load().

3.2.2 Test de positionnement des sommets

Ce test vérifie que tous les sommets se trouvent à une distance cohérente du centre de la planète.
En effet notre planète possède un rayon donné en paramètre dans properties.txt ($planet\_radius$) et une hauteur maximale des montagnes définie aussi dans properties.txt ($height\_max$). On calcule la distance d'un sommet (MeshVertex) avec son attribut $\_position$ et avec la fonction norm() de la bibliothèque Eigen3.
Tous les points doivent donc être positionné à une distance comprise entre l'intervalle [$planet\_radius$, $planet\_radius + height\_max$]. Si c'est le cas, notre test passe.
Les tests passent mais on a du faire une petite modification. Lorsque la position du point était égal à $planet\_radius$, dans certains cas, le test renvoyait faux, alors que la valeur était la même et donc le test aurait du passer.
On a changé le test pour qu'il soit compris dans l'intervalle
|$planet\_radius$ - 0.00001, $planet\_radius + height\_max$].
Dans ce cas, le test passe.

3.2.3 Test de respect de l'intervalle d'humidité

Ce test vérifie que toutes les faces soient compris dans l'intervalle d'humidité qu'il est possible d'avoir. C'est-à-dire entre l'intervalle [0, $MAX\_HUMIDITY$].
Pour cela, on parcoure toutes les faces de la planète à l'aide de l'attribut $\_facesHumidity$ de HumidityFactory, qui contient le numéro de la face, ainsi que son humidité.
Si toutes les faces ont une humidité comprise entre l'intervalle cité précédemment, alors le teste renvoie vrai. Le test a renvoyé vrai donc nos valeurs d'humidités sont bonnes.

3.2.4 Test de connexité des plaques tectoniques

Ce test vérifie que toutes les faces aient au moins une face adjacente appartenant à la même plaque tectonique et donc qu'aucune face se retrouve toute seule.
Pour cela on parcours le surface mesh de la planète ($surface\_mesh$), grâce auquel on récupère l'id des 3 plaques voisines, que l'on compare à l'id de la plaque de la face actuelle.
Si, pour toutes les faces, au moins une des faces voisines appartient à la même plaque alors le test est vrai. Le test est passé donc les faces possèdent au moins un voisin appartenant à la même plaque tectonique.

3.2.5 Test de connexité des zones d'humidité

Ce test vérifie que pour chaque face, son humidité est proche de celle de tous ses voisins.
Pour cela on parcours le surface mesh de la planète, grâce auquel on récupère l'humidité des faces voisines, que l'on compare à l'humidité de la face actuelle. Pour que le test soit valide, il faut que la différence d'humidité entre la face actuelle et les faces voisines soient inférieures au maximum entre les valeurs absolues de ($\_humiditySmoothMin$, $\_humiditySmoothMax$, $\_humidityVariationMin$, $\_humidityVariationMax$). Le test est passé, toutes les faces ont bien des humidités proches à celles de leurs voisins.

3.3 Améliorations et perspectives

3.3.1 Système de sauvegarde

L'implémentation actuelle du système de sauvegarde n'est pas optimale. Pour la sauvegarde d'une planète, le fichier texte est sauvegardé par défaut dans le dossier d'éxécution du programme avec le numéro de la graine comme nom de fichier. Pour ce qui est du chargement d'une sauvegarde, il est actuellement nécessaire de modifier le fichier "properties.txt" qui se trouve dans le dossier d'éxécution du programme et y entrer le numéro de la graine à charger dans le champs *seed*. Il faudrait proposer à l'utilisateur un moyen de choisir le nom du fichier à sauvegarder et un moyen de choisir le fichier à charger, on peut imaginer des paramètres à fournir lors de l'éxecution, des paramètres en ligne de commande ou une interface graphique.

3.3.2 Régénération de la planète

Afin de générer une nouvelle planète, il est actuellement nécessaire de relancer le logiciel. Il faudrait implémenter une interaction permettant de regénérer et d'afficher une planète avec le même paramétrage, voir de permettre de paramétrer la génération directement dans le logiciel.

3.3.3 Relief

L'implémentation actuelle du relief génère aléatoirement des points dans un intervalle autour de la hauteur maximale, or le client a spécifié préférer une implémentation du bruit de Perlin [2].

3.3.4 Amélioration des côtes

Le résultat obtenu par Sebastian Lague (figure 1) génère des planètes avec des côtes plus détaillées que celles obtenues avec ce logiciel, elles possèdent notamment des îles et des péninsules. Il serait possible de repenser notre génération de plaque afin d'obtenir des côtes similaires.

3.3.5 Coloration des océans

Les océans sont de couleur monochrome dans cette version du logiciel, il faudrait ajouter une variation de couleur aux océans afin d'obtenir un rendu plus esthétique. En s'inspirant de la figure 1, on pourrait éclaircir le bleu près des côtes, et assombrir le bleu le plus loin des côtes afin de créer une impression de plage.

3.3.6 Positionnement de la caméra

Si on change le rayon de la planète dans properties.txt (*planet_radius*), la caméra ne s'adapte pas. Si le rayon est trop élevé on se retrouve dans la planète et si le rayon est trop petit on sera trop loin de la planète.
De plus, le zoom et le dézoom sont bloqués par un intervalle. Par exemple, si le rayon est trop élevé et qu'on essaye de dézoomer pour la planète, soit la planète ne s'affiche pas, soit on voit la planète qui est très zoomé, mais on verra jamais la planète en entier.

3.3.7 Génération des rivières

Nous n'avons pas réussi à temps à faire implémenter une génération de rivières fonctionnelle, comme on peut le voir sur la figure ci-dessous.

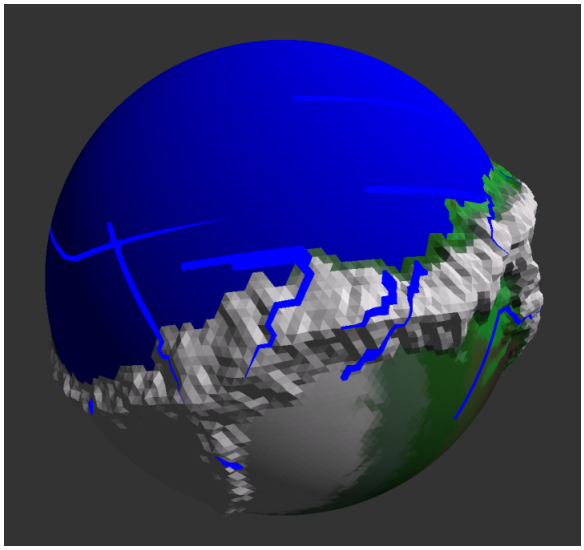

FIGURE 32 – Génération des rivières

4 Conclusion

Le logiciel rendu apporte des réponses aux besoins exprimés par le client, en effet toutes les priorités de niveau 1 et 2 (voir 1.4) sont implémentées, bien que l'une d'entre elle ne soit pas optimale 3.3.1. Un besoin de priorité 3 a également été implémenté indirectement via l'utilisation d'une texture en fonction de l'humidité et la temperature, qui est l'ajout d'un panel de couleur pour les biomes. Plusieurs amélioration sont envisageable à l'avenir, notamment une amélioration de l'ergonomie en apportant une réponse aux problèmes décrit dans la partie précedente, ainsi qu'une amélioration de la documentation et des tests fournis.

5 Lexique

Sont définis ici certains termes essentiels à la compréhension de ce projet :

5.1 Biome

Un biome est une région habitée par différentes espèces végétales et animales partageant des caractéristiques communes. C'est le type de climat d'une région qui défini le type de biome.

5.2 Lancer de Rayon

Le Lancer de rayon(en anglais *Raytracing* est une technique de calcul permettant de modéliser des effets optiques réalistes. Le principe repose la projection de rayon depuis le point de vue (la caméra) afin de récupérer la couleur à la surface d'un objet. Des rayons supplémentaires peuvent être à leur tour être générés afin de calculer des effets de réflexions, de réfractions ainsi que les ombres et les lumières

5.3 Low Poly

Maillage polygonal servant à représenter un objet en 3D comportant un nombre relativement faible de polygones pour des raisons d'optimisation ou de style graphique.

5.4 Caméra

Mot utilisé pour désigner le point de vue à partir duquel on visualise la scène dans le domaine du rendu en 3 dimension.

6 Références et Bibliographie

Références

[1] S. Lague, "[unity] procedural planets," dernier accès 28-01-2020. [Online]. Available : https://www.youtube.com/playlist?list=PLFt_AvWsXl0cONs3T0By4puYy6GM22ko8

[2] "Bruit de perlin," dernier accès 05-04-2020. [Online]. Available : http://sdz.tdct.org/sdz/bruit-de-perlin.html

[3] A. Gainey, "Procedural planet generation," *Experilous*, September 2014, dernier accès 27-01-2020. [Online]. Available : https://experilous.com/1/blog/post/procedural-planet-generation

[4] M. W. Dictionnary, "Tessellation," dernier accès 02-02-2020. [Online]. Available : https://www.merriam-webster.com/dictionary/tessellation

[5] M. Tegmark, "An icosahedron method for pixelizing the celestial sphere," October 1996, dernier accès 27-01-2020. [Online]. Available : https://space.mit.edu/home/tegmark/icosahedron.ps

[6] E. L. A.-M. S. Sylvine Auclair-Semere, Martine Jacquin, "Modélisation à l'aide des diagrammes de voronoï : Réseau téléphonique, et autres applications," dernier accès 28-01-2020. [Online]. Available : http://www.mathom.fr/mathom/sauvageot/Modelisation/Graphes/Voronoi_telephone.pdf

[7] J. Davies, "Random points on a sphere," dernier accès 28-01-2020. [Online]. Available : https://www.jasondavies.com/maps/random-points/

[8] M. Bostok, "Best-candidate circles," dernier accès 28-01-2020. [Online]. Available : https://observablehq.com/@mbostock/best-candidate-circles

[9] K. Perlin, "Java implementation of perlin noise algorithm 2002 version," 2002, dernier accès 28-01-2020. [Online]. Available : https://mrl.nyu.edu/~perlin/noise/

[10] L. University, "Opengl real-time procedural planet rendering," dernier accès 30-01-2020. [Online]. Available : https://www.youtube.com/watch?v=rL8zDgTlXso

[11] S. H. Ahn, "Opengl sphere," dernier accès 30-01-2020. [Online]. Available : http://www.songho.ca/opengl/gl_sphere.html

[12] I. University, "Tectonics plate," dernier accès 01-02-2020. [Online]. Available : http://publish.illinois.edu/platetectonics/

[13] R. Malgouyres, "Programmation 3d c++ opengl glsl," dernier accès 28-01-2020. [Online]. Available : https://malgouyres.org/cours/data/opengl.pdf

[14] Google, "Googletest," dernier accès 02-04-2020. [Online]. Available : https://github.com/google/googletest

[15] "Mondes3d," dernier accès 06-04-2020. [Online]. Available : https://www.labri.fr/perso/pbenard/teaching/mondes3d/td3.html

[16] G. Guennebaud, “Halfedge,” dernier accès 02-04-2020. [Online]. Available : https://www.labri.fr/perso/guenneba/m3d/Cours_Monde3D_2020_COVID19_Halfedge.pdf