Projet de Programmation
Génération procédurale de planètes

Master 1

Professeur accompagnant: M. X
Cahier des besoins rédigé par:
Alexey Zhukov, Tony Wolff, Baptiste Bedouret,
Alexis Marec, Antoine Fredefon, Thomas Mercier

02/03/2022

Contents

1 Introduction et objectifs du projet **3**

2 Analyse de l'existant **4**
2.1 Bibliothèque de génération de bruit 4
2.2 Algorithmes de niveau de détail 6

3 Description des besoins **8**
3.1 Génération de heightmaps 8
3.2 Niveau de détail 9
3.3 Visualisation OpenGL 10
3.4 Besoins Fonctionnels 11
3.5 Besoins Non-Fonctionnels 12

4 Les scénarios d'utilisation **13**
4.1 Scénario : 13
4.2 Extensions : 13

5 Tests **14**
5.1 Test Génération des heightmaps 14
5.2 Test Algorithme CDLOD 14
5.2.1 Génération d'un quadtree à partir d'une heightmap 14
5.2.2 Sélection des noeuds pour le rendu 14
5.3 Test des interactions utilisateur 15
5.3.1 Zoom 15
5.3.2 Activer le maillage 15
5.3.3 Raccourcis clavier 15

1. Introduction et objectifs du projet

La génération procédurale de planètes est un problème complexe qui requiert des compétences dans des domaines variés, notamment pour modéliser la sphère avec des niveaux de détails variables en fonction de la résolution demandée. Dans le cadre de ce projet, la carte de la planète sera générée au vol par un ou plusieurs algorithmes conseillés par le sujet. Notre première étape consiste donc à trouver, comprendre et utiliser une bibliothèque de bruit de Perlin ou bruit de Simplex pour générer une heightmap. La principale difficulté de cette étape sera de génerer des cartes sphériques.

Dans un second temps, notre objectif sera de créer une bibliothèque permettant le stockage et la génération de la heightmap d'une planète (carte de hauteur représentant les variations du terrain) utilisant les principes de niveaux de détails. Les algorithmes et les bibliothèques que nous allons étudier puis mettre en oeuvre servent à modéliser ces heightmaps de manière efficace, mais aussi à réguler la quantité de sommets à tracer lors du rendu graphique.

Enfin, dans le but de visualiser la planète, nous devrons développer un outil de visualisation simple utilisant la bibliothèque OpenGL. En plus de donner un aperçu de la planète (et donc de la qualité de la heightmap générée), cet outil permettra à l'utilisateur de se déplacer dans la scène pour faire apparaître les différents niveaux de détail du maillage.

Dans l'ensemble, cet outil permettra de générer la carte de hauteur d'une planète de manière procédurale, de la stocker dans une structure adaptée pour effectuer du LOD, et enfin de la visualiser dans une fenêtre OpenGL comme nous pouvons le voir sur la figure 1.1.

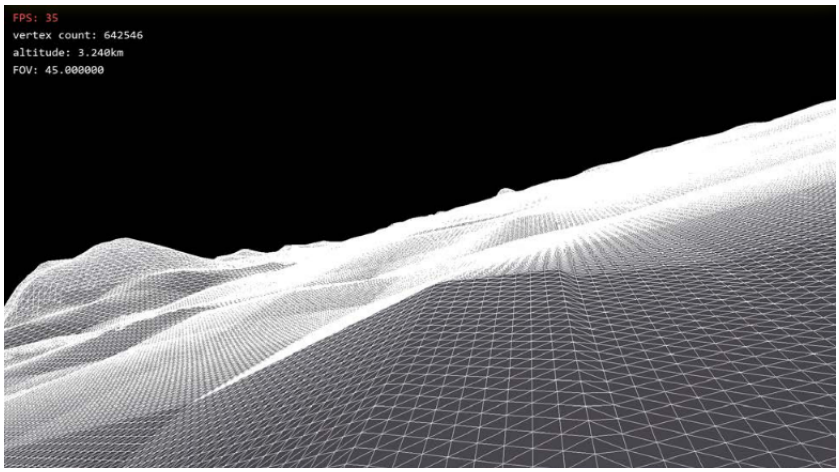

Figure 1.1: Exemple de visualisation de la surface de la lune

2. Analyse de l'existant

2.1 Bibliothèque de génération de bruit

Une bibliothèque de génération de bruit sera utile pour la création d'une heightmap 2D du moins. Définisson le terme : Une carte de hauteur est une image 2D monochrome. La valeur du pixel s'interprète comme la distance du terrain par rapport au sol, une valeur élevée se traduit par du blanc sur l'image, une valeur basse par des nuances de noirs, le noir total étant le niveau de la mer. Ces cartes sont créés soit à la main par des artistes, soit par des données de cartes déjà existantes, ou bien par des algorithmes de génération de bruit (notre solution).

1. Libnoise Bibliothèque C++ de production de bruit "cohérent", bruit à variation régulière (classe de bruit dont font partie Perlin et Simplex). Comme indiqué sur la page web [Bev], elle a déjà été utilisée sciemment dans le cas de la création d'une carte 2D pour une planète quelconque.

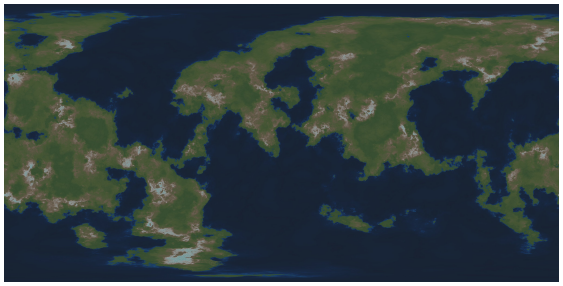

Figure 2.1: texture d'une planète

Elle précise bien que ce n'est pas un outil de rendu et qu'il faudra utiliser une autre bibliothèque ou notre propre code pour générer une image, elle propose l'utilisation de **noiseutils** qui sauve le désagrément décrire des classes telles que le remplissage de noise map (tableau 2D qui a vocation à recevoir les valeurs générées par les modules de bruit), des builders associés à ces noise map pour des objets mathématiques utiles comme la sphère dans notre cas, des classes pour sérialiser une image ou une noise map, et des classes pour créer des images bien entendu.
D'après la documentation, un bruit "cohérent" respecte trois critères :

   (a) Si l'on passe la même valeur d'entrée, on obtient toujours la même valeur de sortie.

   (b) Un petit changement dans la valeur d'entrée produira un petit changement dans la valeur de sortie.

   (c) Un changement important de la valeur d'entrée produira un changement aléatoire de la valeur de sortie.

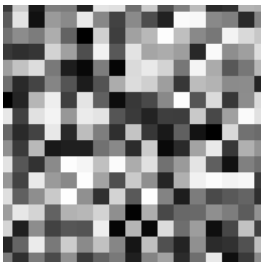

Figure 2.2: texture 2D obtenu à l'aide de la fonction continue de bruit cohérent

Elle part d'une fonction pseudo-random tel que *rand()*. en C++ et prend en paramètres des entiers (ou réel pour la version continue), et renvoyant une entier entre -1 et 1, c'est *integerNoise*.

Pour éviter les froissements dans la texture, l'interpolation des valeurs entières se présente comme immanquable pour avoir des transitions douces entre valeurs de bruit. L'interpolation linéaire est une première version simple mais ne suffit pas à créer une niveau de détail naturel, alors la fonction de bruit cohérent va chercher à utiliser une version non linéaire, couplé à l'interpolation de vecteurs gradient à la place des valeurs entières de bruit. Le vecteur gradient est obtenu avec *integerNoise* qui le sélectionne aléatoirement dans un ensemble de vecteurs précalculés. On utilisera la version 2D de cette fonction, chaque dimension correspond à un paramètre (2 dimensions égal à deux paramètres). Note : un ensemble de fonctions de bruits "cohérents" sera indispensable pour avoir l'essence d'une texture terrain, cette ensemble de fonction se retrouve dans le bruit de Perlin ou de Simplex.

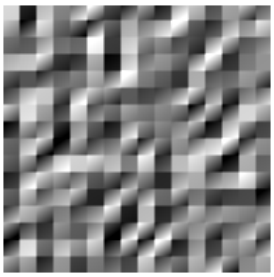

Figure 2.3: bumpmap "froissée"

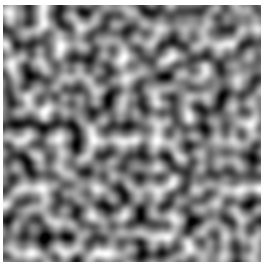

Figure 2.4: texture 2D avec vecteur gradient

2.2 Algorithmes de niveau de détail

Étant donnée l'importance de l'optimisation pour le rendu temps-réel, il existe de nombreuses approches abordant le problème du stockage et du rendu de heightmaps. La technique la plus couramment utilisée à l'heure actuelle se base sur des algorithmes de niveaux de détail (Level Of Detail ou LOD) pour subdiviser le maillage de la sphère en fonction de la distance au sol de l'observateur.

Dans un premier temps, nous avons étudié l'article de Filip Strugar *Continuous Distance-Dependent Level of Detail for Rendering Heightmaps*[**Str10**]. Bien que paru en 2010, cet article permet de comprendre plus en détail comment représenter une heightmap sous forme de quadtree (arbre quaternaire dont les particularités sont expliquées plus bas) et ainsi utiliser cette structure de données pour créer l'algorithme de rendu de heightmaps CDLOD.

Un quadtree ou arbre quaternaire est un arbre dont chaque noeud dispose au maximum de quatre enfants. Cette représentation est particulièrement utile pour effectuer une subdivision d'un espace en deux dimensions. En effet, une image 2D par exemple peut être partitionnée en quatre quadrants de tailles égales, comprenant chacun l'information de ce quart d'image. Puis, chaque quadrant peut être lui aussi subdivisé en quatre, et ainsi de suite tant que des éléments peuvent être distingués dans celui-ci. Un bon moyen de visualiser le processus de création d'un quadtree est l'application interactive **quadtreevis**[1], permettant de visualiser la division de l'image de base ainsi que les noeuds de l'arbre associé.

Pour appliquer ce principe au stockage de heightmap, l'algorithme CDLOD pose la contrainte suivante : La profondeur à laquelle nous nous trouvons dans le quadtree correspond toujours au niveau de détail. En d'autres termes, le noeud le plus élevé de l'arbre correspond au niveau de détail le plus bas, et chaque fils comprendra quatre fois plus d'informations (ici des triangles) que les noeuds de l'étage précédent. La figure suivante, tirée de l'article, représente la division d'une heightmap en 4 niveaux de détail : Du moins détaillé (LOD 3) au plus détaillé (LOD 0).

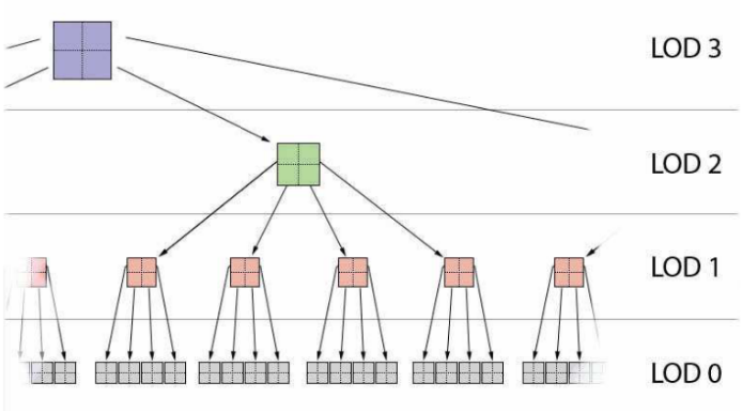

Figure 2.5: Niveaux de détails sur un quadtree

[1] https://jimkang.com/quadtreevis/

Ainsi, l'algorithme pourra simplement sélectionner les noeuds correspondant au bon niveau de détail en fonction de la position relative de l'utilisateur par rapport au maillage. Notamment, en utilisant la distance réelle entre l'observateur et les sommets du maillage, il est possible de représenter un nombre de points constant à l'écran et rendre l'algorithme plus prévisible, en théorie. Bien sûr, ces méthodes doivent être adaptées au fait que nous travaillons sur une surface sphérique et non plane, ce qui peut avoir des conséquences sur le calcul de distances et donc sur le rendu final.

Selon l'auteur, cette méthode permet de répondre à quelques défaillances des algorithmes de LOD classiques, et met en évidence des besoins qui pourront nous être utiles :

- En utilisant la distance réelle et non simplement la latitude/longitude de l'observateur, l'algorithme apporte plus de précision sur les niveaux de détail à afficher. Cet aspect est particulièrement important dans le cas d'une surface sphérique comme c'est le cas ici.
- Il permet d'éviter l'apparition d'artefacts graphiques lors des transitions, car le maillage est complètement remplacé avant l'étape de transition.
- En terme de performances : Les algorithmes classiques requièrent des calculs additionnels pour afficher des transitions fluides entre les différents niveaux de détail en créant des connexions (sommets) intermédiaires, ce qui n'est pas le cas pour l'algorithme CDLOD.

Au final, nous aurons à chaque frame du rendu une carte divisée en niveaux de détails dépendant de la position de l'utilisateur. Sur la figure 2.6, on remarque bien que la division s'affine d'un facteur 2 plus on se rapproche de l'observateur. La zone sombre représente les parties en dehors du champs de vision.

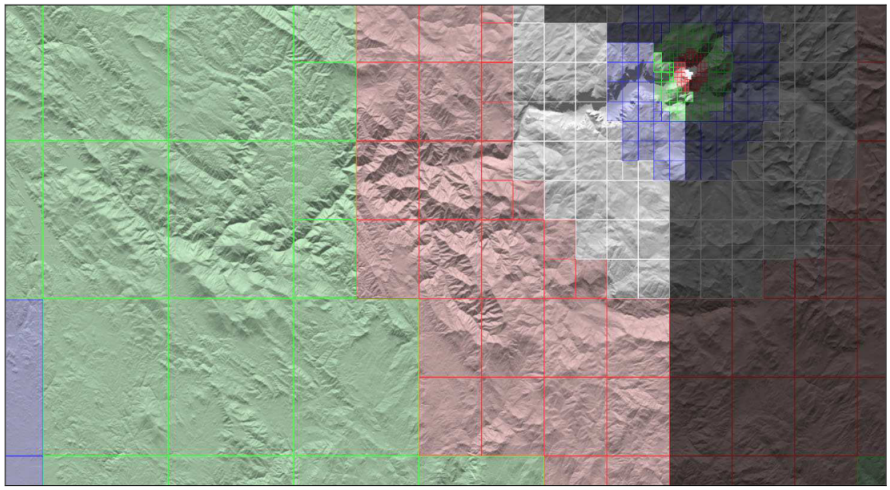

Figure 2.6: carte découpée en LOD

Bibliothèque de gestion de quadtrees : https://github.com/dfriend21/quadtree

3. Description des besoins

3.1 Génération de heightmaps

Besoins fonctionnels

Besoins non-fonctionnels

3.2 Niveau de détail

Besoins Fonctionnels

1. **Lecture des heightmaps en entrée** : De manière générale, les heightmaps sont stockées dans des formats d'images standard car elles représentent une structure de valeurs en 2D, chaque pixel contenant la hauteur du terrain en ce point. Dans un premier temps, et dans un soucis de simplification, l'idéal serait de récupérer un fichier RAW contenant les données brutes de la heightmap codées en 16 ou 32 bits selon la précision nécessaire. Par la suite, nous pourrons étendre l'éventail des formats acceptés.

2. **Algorithme CDLOD** :

   (a) La première étape de l'algorithme est le stockage de la heightmap sous forme de quadtree. Cette étape implique l'utilisation ou la création d'une bibliothèque de gestion de quadtrees. Celle-ci devra permettre au minimum la création et la suppression d'un arbre, l'ajout et la suppression de noeuds ainsi qu'une méthode de parcours de l'arbre.

   (b) Dans un second temps, nous devons effectuer la sélection des noeuds du quadtree pour le rendu. Cette action est effectuée à chaque fois que l'observateur se déplace dans la scène, et donc possiblement à chaque frame. Pour savoir quels noeuds sont sélectionnés, les distances couvertes par chaque niveau de LOD sont pré-calculées (cf Partie 2.2 : *Algorithmes de niveau de détail*). A chaque fois qu'un mouvement est détecté, l'algorithme recherche les noeuds représentant les parties du terrain actuellement visibles et le niveau de détail avec lequel elles doivent être tracées.

   (c) Enfin, chaque noeud sélectionné doit être stocké dans une structure de données temporaire qui doit contenir sa position dans la scène, sa taille et son niveau de détail. D'autres informations peuvent y être ajoutées si elles sont nécessaires au rendu graphique.

3. **Communication avec le module de rendu** : Comme nous pouvons le constater, les modules vont échanger une grande quantité d'informations et ce possiblement à chaque frame du rendu.

Besoins non-fonctionnels

Taille des données en mémoire : L'implémentation pratique d'un logiciel de rendu de terrain doit tenir compte de la taille de ce dernier. Ici, il y a deux facteurs principaux à prendre en compte :

1. **Les données de terrain** : Pour ne pas garder la totalité des données de la heightmap en mémoire, celle-ci peut-être divisée en blocks représentants les morceaux de terrain potentiellement visibles par l'observateur. Ces fragments peuvent être transmis sous forme de data-stream pendant le rendu. Cette méthode est d'autant plus efficace que l'algorithme CDLOD effectue déjà une sélection des blocks lors du calcul de LOD.

2. **Les données du quadtree** : Plus volumineux encore, les noeuds du quadtree contiennent des détails tels que la taille, la position, les voisins du noeud, etc... beaucoup d'informations qui peuvent être superflues pour l'étape de rendu. La version StreamingCDLOD de l'algorithme présenté dans le chapitre 2.2 évite ce problème en ne gardant que les valeurs minimales et maximales de la surface couverte par le noeud, qui sont stockées dans une matrice 2x2 pour chaque noeud. Le reste des données est automatiquement généré lors du parcours du quadtree.

Enfin, il est possible de compresser d'une part les heightmaps, et d'autre part les données des noeuds les plus détaillés du quadtree, en stockant les valeurs de la matrice min/max du noeud dans l'espace inutilisé de la matrice du noeud parent avec une faible perte de précision.

3.3 Visualisation OpenGL

Besoins fonctionnels

1. **Créer une boîte à outils pour travailler avec quadtree** : Un nœud dans un quadtree est un plan simple à l'intérieur duquel se trouve un certain nombre de sommets. Si ce plan croise notre pyramide de vue, il est nécessaire de rendre les sommets à l'intérieur de ce plan. Si un plan a des nœuds enfants à l'intérieur, il ne contient pas de sommets. Les sommets contiennent soit ses nœuds enfants, soit à leur tour leurs nœuds enfants, etc. cette récursivité peut durer aussi longtemps que nous atteignons la valeur maximale de la récursivité, ou le nombre de triangles ne devient pas inférieur au nombre maximum de triangles pour chaque nœud. Après avoir atteint ces valeurs, le nœud arrête la division.
2. **Frustum Culling (Rendu uniquement des polygones visibles)** : Frustum est une pyramide de vue. Son sommet est la position de la caméra. Ses plans passent à travers les côtés de l'écran du moniteur. Tout ce qui ne tombe pas dans cette pyramide n'a pas â être dessiné car les sommets ne sont pas dans le champs de vision de la caméra.
3. **Créer un objet volumétrique (sphère) avec une carte heightmap** : Étant donné que la planète est utilisée comme objet, il faut créer une fonction pour réaliser un placage sphérique.
4. **Intéraction utilisateur** :
   (a) Pouvoir utiliser les flèches du clavier pour se déplacer dans la scène.
   (b) Pouvoir utiliser la souris pour zoomer et appliquer une rotation à la planète.
   (c) Ajouter un menu ou des raccourcis claviers pour les options telles que : quitter, mettre en pause ou encore afficher des informations système (nombre de sommets, Images par secondes)

Besoins non-fonctionnels

1. **Optimisation du quadtree :**

   (a) Le nombre maximal de triangles dans un nœud peut dépendre de la taille de la scène. Si elle est trop petite, le programme peut ralentir, car il est nécessaire de traiter des milliers de nœuds. Il est optimal de commencer à environ 1000, puis de changer la valeur jusqu'à ce qu'il y ait une meilleure option.

3.4 Besoins Fonctionnels

1. Générer une technique de rendu de heightmap d'une surface:

   (a) Dans ce cas utiliser l'algorithme CDLOD pour pouvoir garder un niveau de subdivisions du maillage (quadtree) de la sphère à un niveau de détail qui se régule en fonction de la distance au sol. En effet, on utilise un quadtree qui va créer un sous ensemble de noeuds affichables à l'écran et va parcourir l'arbre du niveau de détail le moins élevé au plus élevé. Les données des noeuds sont utiles au rendu de la parcelle de terrain visible.

   (b) Implémenter et tester plusieurs calculs de distance pour s'adapter à la surface sphérique.

   (c) Nombre constant de sommets affichés à l'écran: calcul de la surface ainsi que du nombre de sommets que chaque couche LODS couvre.

   (d) Utiliser une bibliothèque de production de bruit pour ainsi pouvoir générer la texture d'une surface. Utiliser par exemple un bruit de Perlin.

3.5 Besoins Non-Fonctionnels

a. Des quantifications:

- Taille :
- Performance :
  - Passage à l'échelle : Lors de la création du quadtree ou dans la phase de rendu, s'assurer que le programme puisse gérer des heightmaps très grandes et très détaillées.
  - Garder un affichage fluide de rendu de heightmap.
  - Le temps de traitement du programme ne dépasse pas quelques minutes.
  - Récupérer la taille totale et restante de la mémoire du GPU avec GL NVX gpu memory info.
- Facilité d'utilisation :
  - Fenêtre graphique dans laquelle se trouve le nombre de sommet que compte l'image et l'altitude.
- Contraintes et difficulté techniques:
  - Surface sphérique. L'ordinateur doit générer un maillage de base convexe pendant la génération du terrain.
  - Contraintes d'affichage: Faire des tests pour pouvoir afficher un nombre conséquents de triangles.
  - L'ordinateur doit perdre le moins de temps possible à rendre le terrain et les sommets du maillage qui ne sont pas visibles par la caméra. On utilise donc pour cela une méthode nommée "culling" qui n'affiche que la partie qui vue par la caméra. La partie en dehors n'est pas affichée. Cela permet de réduire le temps de calcul.
  - Il est obligatoire d'utiliser une machine possédant une carte graphique
  - Vérifier la version d'openGL. Utiliser de préférence une openGL 3.3.
- Portabilité:
  - Utiliser une machine contenant une carte graphique avec une version d'opengl supérieur à ..
- L'énonciation de risques et parades:
  - Utiliser les bons algorithmes qui utilise bien le GPU.
  - Utiliser un maillage de triangle équilatéraux plutôt que d'autres polygones.
- Fiabilité sécurité:
  - Enregistrer et push le travail avec git.

4. Les scénarios d'utilisation

4.1 Scénario :

1. Le client lance le fichier exécutable
2. Un explorateur de fichier s'ouvre, demandant à l'utilisateur de spécifier l'emplacement d'une heightmap. Le logiciel va alors stocker cette heightmap sous forme de quadtree.
3. Une fenêtre graphique s'ouvre affichant le rendu placé sur le centre de la heighmap.
4. Le client dezoom via la molette de sa souris.
5. Le client se déplace sur la surface à l'aide des flèches de son clavier.
6. Le client applique une rotation sur la planète en appuyant sur la touche R.
7. L'utilisateur choisit de changer de heightmap en appuyant sur la touche C.
8. L'utilisateur active le maillage en appuyant sur la touche M.
9. Le client choisit d'afficher des informations sur le programme en appuyant sur la touche H.

11. L'utilisateur met le programme en pause en appuyant sur P.

10. Le client quitte le logiciel en appuyant sur Q.

4.2 Extensions :

1a. La configuration de l'appareil du client ne permet pas de faire fonctionner notre logiciel.

- Un message d'erreur s'affiche et le logiciel ne s'exécute pas.

2a. L'utilisateur ferme l'explorateur de fichier

- Le logiciel se ferme

2b. Le client utilise le logiciel avec un fichier ne correspondant pas à une heightmap ou pas compatible avec notre implémentation.

- Un message d'erreur s'affiche et le logiciel se ferme.

5. Tests

5.1 Test Génération des heightmaps

1. Description : Vérifier que la heightmap générer est valide et pourra être lu par notre logiciel
2. Données utilisées :
   (a) Fichier vide
   (b) Fichier non vide mais invalide
   (c) Fichier non vide mais d'un type incorrect
   (d) Fichier non vide de type correct mais avec des données corrompues
   (e) Fichier correct, on pourra par exemple le générer avec la bibliothèque [Bev]

   Résultats attendus : Si le fichier est invalide le test doit échoué et afficher un message d'erreur contenant une description du problème.

5.2 Test Algorithme CDLOD

5.2.1 Génération d'un quadtree à partir d'une heightmap

1. Description : Vérifier que le quadtree générer à partir d'une heightmap est valide, c'est à dire qu'il respecte la définition d'un quadtree.
2. Données utilisées :
   (a) Heightmap vide
   (b) Heightmap non vide
3. Résultats attendus : Si la heightmap est vide alors le Quadtree doit l'être aussi, dans le cas contraire on vérifiera la validité du Quadtree généré

5.2.2 Sélection des noeuds pour le rendu

1. Description : Vérifier que la sélection des noeuds du quadtree se réalise correctement
2. Données utilisées :
   (a) Quadtree vide
   (b) Quadtree non vide

3. Résultats attendus : Si le quadtree est vide alors l'algorithme ne doit pas essayer de sélectionner de noeuds dans le cas contraire CDLOD doit nous renvoyez les noeuds correspondants et leurs niveau de détails

5.3 Test des interactions utilisateur

5.3.1 Zoom

1. Description : Vérifier qu'un zoom de la part de l'utilisateur, zoom sur la planète et change le niveau de détail.
2. Déroulement du test : Une fois que l'utilisateur a zoomé l'algorithme CDLOD doit nous envoyez les noeuds représentant les parties du terrains visibles et la fenêtre graphique doit se mettre à jour.

5.3.2 Activer le maillage

1. Description : Vérifier que si l'utilisateur active le maillage, celui-ci s'affiche à l'écran
2. Déroulement du test : Une fois que l'utilisateur a activé le maillage, l'algorithme CDLOD doit nous envoyez les informations du quadtree courant : Sommets et arrêtes et la fenêtre graphique doit se mettre à jour.

5.3.3 Raccourcis clavier

1. Description : Vérifier qu'à chaque saisie d'un raccourci clavier pour se déplacer la fenêtre graphique se met à jour
2. Déroulement du test : Une fois que l'utilisateur a saisi le raccourci clavier, on vérifie que la position de la caméra soit modifier en accord avec les paramètres définis.

Bibliography

[Bev] Jason Bevins. Complex planetary surface.

[Str10] Filip Strugar. Continuous distance-dependent level of detail for rendering heightmaps. *Journal of Graphics GPU and Game Tools*, July 2010. url : https://github.com/fstrugar/CDLOD/blob/master/cdlod_paper_latest.pdf.