Projet de Programmation
Génération procédurale de planètes

Master 1

Professeur accompagnant: M. Mansencal
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

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 hauteur de la planète sera générée au vol par un ou plusieurs algorithmes conseillés par le sujet. Cette carte, couramment appellée heightmap, représente en chaque point la distance séparant un sommet et le centre de la planète. Notre première étape consiste donc à trouver, comprendre et utiliser une bibliothèque pour générer des heightmaps de manière procédurale. La principale difficulté de cette étape sera de génerer des cartes sphériques simulant la surface d'une planète.

Dans un second temps, notre objectif sera de créer une bibliothèque permettant le stockage de la heightmap d'une planète en utilisant les principes de niveau de détail (Level of Detail ou LOD). En effet, les algorithmes de LOD permettent de sauvegarder les cartes dans différents niveaux de précision que nous pouvons contrôler. Les algorithmes que nous allons étudier puis mettre en oeuvre permettront alors de stocker la heightmap de manière optimale pour faire du LOD, c'est à dire réguler la quantité de sommets à tracer selon la distance entre le maillage et l'observateur.

Enfin, dans le but d'afficher 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 de la surface.

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 et 1.2.

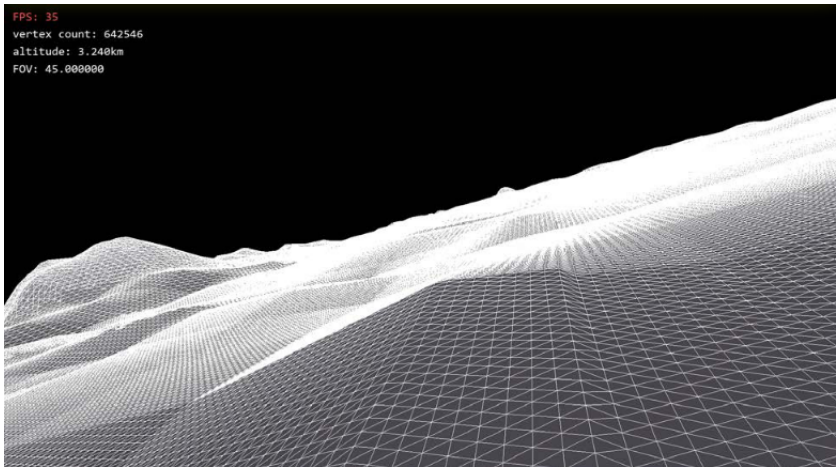


Figure 1.1: Exemple de visualisation de la surface de la lune https://dept-info.labri.fr/ narbel/PdP/Subjects21-22/CLOD-Planets/Planet$_R$*endering.pdf*

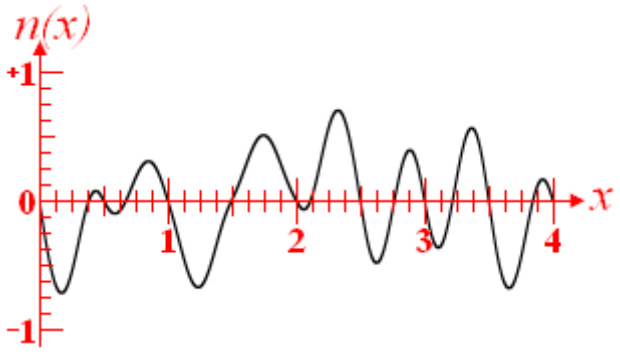
Figure 1.2: Exemple de visualisation de la surface de la lune https://dept-info.labri.fr/ narbel/PdP/Subjects21-22/CLOD-Planets/Planet$_R$*endering.pdf*

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. Définissons le terme : Une carte de hauteur est une image 2D souvent 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). Plusieurs méthodes sont disponibles pour produire des heightmaps. Elles s'obtiennent à partir d'algorithmes de bruits cohérents tels que le bruit de Perlin, le bruit de Simplex, l'algorithme du déplacement du point médian etc. Simplex est une tentative d'évolution de Perlin, l'algorithme réduit le nombre d'opérations et facilite le travail dans les dimensions supérieures. Les bénéfices sont à considérer lorsque nous avons de grandes dimensions, et l'algorithme étant breveté, nous choisirons de travailler avec Perlin Noise. Le déplacement du point médian ne sera pas exploré pour ce projet, je juge le temps insuffisant pour implémenter l'algorithme, de plus la bibliothèque retenue ne l'utilise pas, il est néanmoins intéressant de savoir que l'algorithme a une complexité mémoire plus élevé que Perlin, il nécessite l'enregistrement des calculs intermédiaires.

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. Libnoise génère ce qu'on appelle du bruit cohérent, un type de bruit pseudo-aléatoire régulier, opposé aux bruits purs.



http://libnoise.sourceforge.net/examples/complexplanet/index.html

Figure 2.1: Exemple de bruit cohérent

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.

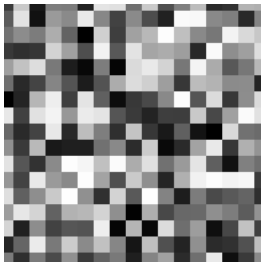
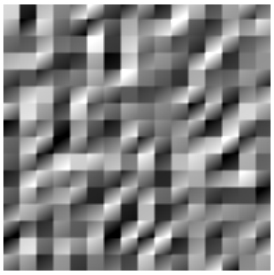
 http://libnoise.sourceforge.net/noisegen/index.html

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*.

Le froissement ou plissage de texture [Figure 2.4] est une imperfection visuelle sur une texture 2D, ces artéfacts forment un pincement à certains endroits. Ils apparaissent car la dérivé de la fonction linéaire aux bornes entières n'est pas définie. 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.



http://libnoise.sourceforge.net/noisegen/index.html

Figure 2.3: bumpmap "froissée"

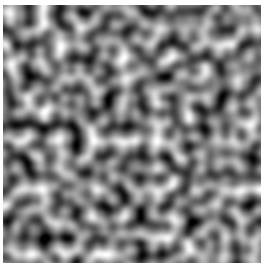


Figure 2.4: texture 2D avec vecteur gradient

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. Le critère sera le nombre maximum de triangles autorisés dans un quadrant . La valeur maximale sera fixée à 10 000, de sorte que chaque quadrilatère continuera à être divisé jusqu'à ce qu'il contienne moins de 10 000 triangles. 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.

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

2.2.1 Description de l'algorithme quadtree

Dans ce projet nous allons utiliser cet algorithme pour pouvoir afficher un niveau de subdivisions du maillage de la surface. Sur ce site web [2] nous avons trouvé un exemple de code détaillé sur l'algorithme quadtree qui explique comment créer cela.

Pour illustrer le fonctionnement de l'algorithme de l'arbre, nous commençons par un quadruple qui englobe la totalité du terrain. Puis nous le divisons en quatre quadrants et vérifions si chacun de ces quadrants contient plus de 10 000 triangles ou non. Si le quadrilatère contient plus de 10 000 triangles, nous le divisons en quatre quadrilatères de taille égale et nous vérifions à nouveau si chacun des quatre nouveaux quadrilatères contient moins de 10 000 triangles ou non. Ainsi de suite. Une fois que chaque quadrilatère de l'arbre entier contient moins de 10 000 triangles, nous avons fini de diviser le terrain en sections. Maintenant, pour chaque quadrant, nous déterminons quels triangles lui appartiennent. Nous pouvons représenter cela sous forme de diagramme. Voir figure 2.5. L'algorithme utilise le noeud supérieur de l'arbre pour représenter l'ensemble du terrain. Le deuxième niveau représente les quatre premiers quadrants ainsi de suite jusqu'à ce qu'il réponde aux critères de division ou non de chaque quadrant.

Par la suite,on crée une classe pour générer un quadtree et on utilise une structure qui prend en charge le stockage et le rendu des informations sur les sommets du terrain. Ensuite, on génère une structure de données utilisée pour la gestion du terrain procédural. Chaque nœud de l'arbre quadruple sera défini comme suit : position, taille, nombre de triangles, tampons et quatre nœuds enfants. La classe aura notamment besoin d'une liste des sommets et du noeud parent pour construire le quadtree de manière récursive.

2.2.2 Description de l'algorithme CDLOD

Pour appliquer ce principe au stockage de heightmap, on utilise l'algorithme CDLOD qui 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).

[2] https://www.rastertek.com/tertut05.html

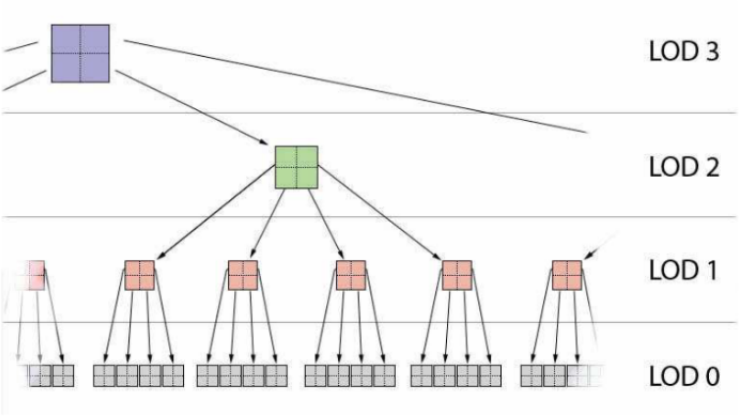


Figure 2.5: Niveaux de détails sur un quadtree
https://dept-info.labri.fr/ narbel/PdP/Subjects21-22/CLOD-Planets/cdlod$_l$*atest.pdf*

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.

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

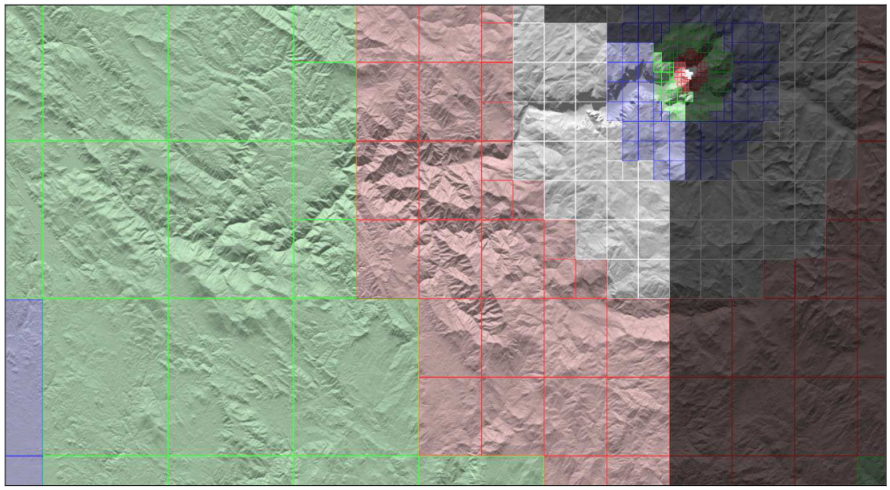


Figure 2.6: carte découpée en LOD
https://dept-info.labri.fr/ narbel/PdP/Subjects21-22/CLOD-Planets/cdlod$_l$*atest.pdf*

3. Description des besoins

3.1 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.

4. **Visualisation OpenGL** :

   (a) **Fenêtre de visualisation** : Création par programmation d'une fenêtre de visualisation qui interagit avec un objet de heightmap. La première étape pour utiliser OpenGL est de créer un context et généralement une fenêtre associée capable d'afficher un rendu OpenGL. Ces deux tâches seront effectuées par la bibliothèque GLFW.

- **GLFW** est une bibliothèque Open Source multiplateforme pour le développement OpenGL sur le bureau. Il fournit une API simple pour créer des fenêtres, des contextes et des surfaces, recevoir des entrées et des événements.

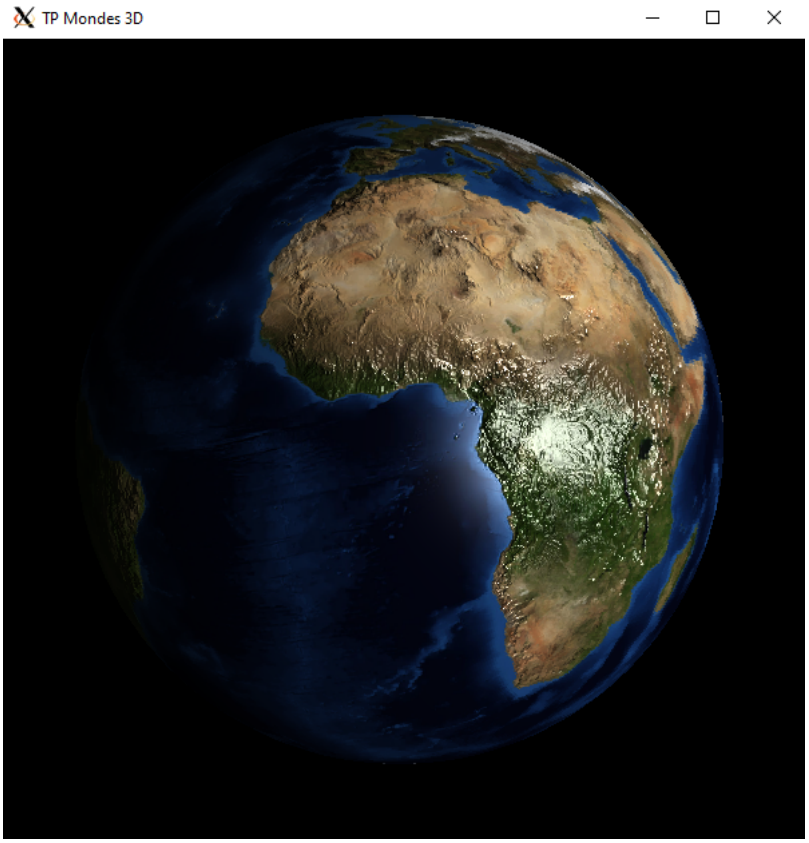


Figure 3.1: Exemple de fenêtre de visualisation. WSL2. XLaunch.

(b) **Création de la fonctionnalité nécessaire et intégration de l'existant pour un fonctionnement pratique de base** : Dans cette partie, nous allons utiliser deux bibliothèques : Eigen et Glbinding.

- **Eigen** est une bibliothèque C++ d'algèbre linéaire. Elle sera utilisée pour la représentation et manipulation de matrices, vecteurs, transformations géométriques et résolution de systèmes d'équations.
- **Glbinding** exploite des fonctionnalités C++ 11 telles que les classes enum, les lambdas et les modèles variadiques, au lieu de s'appuyer sur des macros ; tous les symboles OpenGL sont de véritables fonctions et variables. Il fournit des paramètres de sécurité de type, des en-têtes d'API par fonctionnalité, une résolution de fonction paresseuse, une prise en charge multi-contextes et multi-threads, des rappels de fonctions globaux et locaux, des méta-informations sur la liaison OpenGL générée et le runtime OpenGL, ainsi que des outils et des exemples.

(c) **Initialisation de la caméra** : La portée de vue de la caméra et sa position décident des nœuds à dessiner et de la profondeur de la cascade des nœuds. Les nœuds les plus proches de la caméra sont en cascade jusqu'aux nœuds feuilles, dessinant une géométrie plus dense. La fonctionnalité de la caméra consiste à déplacer l'utilisateur sur une zone ou à modifier la distance de

rendu d'un objet à l'aide des boutons.

- PAGE UP : S'éloigner de l'objet
- PAGE DOWN : Se rapprocher de l'objet
- KEY UP : Tourner la caméra vers le haut
- KEY DOWN : Tourner la caméra vers le bas
- KEY LEFT : Tourner la caméra à gauche
- KEY RIGHT : Tourner la caméra à droite
- KEY W : définir le mode wireframe
- KEY E : activer / désactiver les mesh edges
- KEY + et KEY - : pour définir la plage de vues quadtree

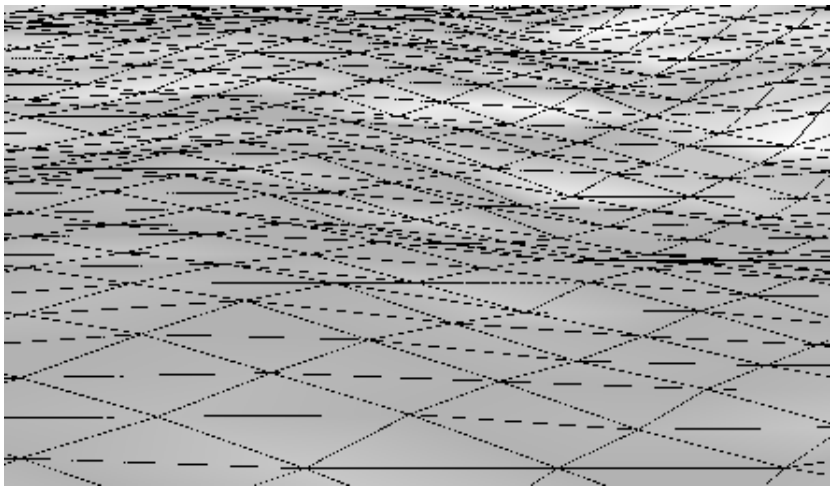


Figure 3.2: Exemple de mesh edges.
https://github.com/drecuk/QuadtreeTerrain

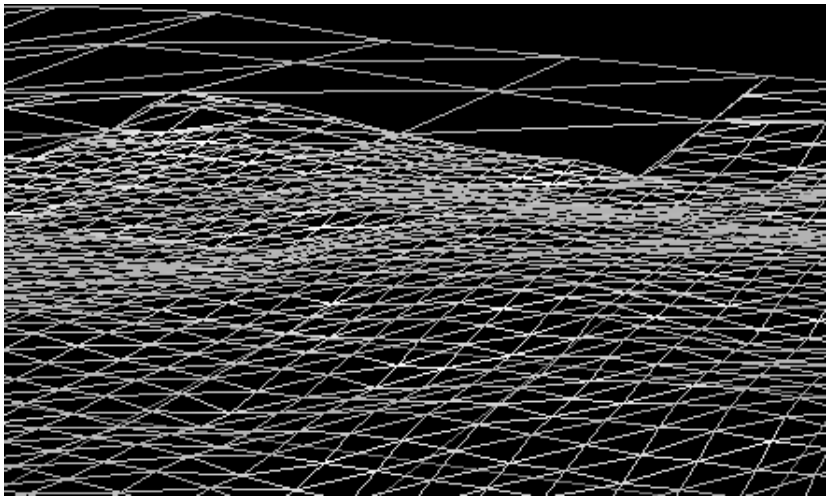


Figure 3.3: Exemple de visualisation d'un Quadtree en mode wireframe
https://github.com/drecuk/QuadtreeTerrain

(d) **Créer une boîte à outils pour travailler avec des objets** : Après avoir traité le heightmap, il doit être installé sur la sphère. Les bibliothèques ObjFormat et tinyobjloader seront utilisées pour cela.

- **tinyobjloader et ObjFormat** sont des bibliothèques pour charger des maillages au format Wavefront OBJ.

(e) **Chargement du RAW bitmap** : Le remplissage de la structure de données avec les données RAW bitmap de 8 bits pour chaque élément du tableau. Ce sont ces données qui seront utilisées pour visualiser le heightmap.

(f) **Implémentation logicielle de CDLOD** : La supclasse crée et maintient une structure quadtree. Chaque noeud de cette structure est décrit par la structure de noeud. ID et ParentID associent le noeud actuel au parent, et l'index de branche enregistre les ID des nœuds enfants. Ceci décrit les points à partir desquels les quads OpenGL sont construits (surface du terrain). Chaque nœud a 3 x 3 sommets, que notre constructeur utilise pour dessiner des triangles.

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. Au lieu d'envoyer toutes les informations à votre GPU, on trie les éléments visibles et invisibles et on ne rend que les éléments visibles. Grâce à cette technique, on gagne du temps de calcul GPU. 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.
3. **Performance**: Le temps de traitement du programme ne dépasse pas quelques minutes.
4. **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.
5. **Visualisation OpenGL** :

(a) **Performance:**

i. Le temps de traitement du programme ne dépasse pas quelques minutes.

ii. Garder un affichage fluide de rendu de heightmap.

(b) **Contraintes et difficulté techniques:**

i. Surface sphérique. L'ordinateur doit générer un maillage de base convexe pendant la génération du terrain.

ii. Utiliser un maillage de triangle équilatéraux plutôt que d'autres polygones.

(c) **Portabilité:**

i. Utiliser une machine contenant une carte graphique avec une version d'openGL de préférence openGL 3.3.

ii. Ajouter le contrôle de la molette de la souris au lieu de boutons, pour un contrôle plus facile de la caméra.

(d) **Contraintes d'affichage:**

i. Faire des tests pour pouvoir afficher un nombre conséquents de triangles.

ii. Récupérer la taille totale et restante de la mémoire du GPU avec GL NVX gpu memory info.

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.

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.
10. L'utilisateur met le programme en pause en appuyant sur P.
11. 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 générée est valide et pourra être lue 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 la validité du quadtree suivant sa définition.
2. Données utilisées :
   (a) Heightmap vide
   (b) Heightmap non vide
   (c) Autre fichier
   (d) Heightmap corrompue
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 : Opérer un zoom, observer le changement du niveau de détail (ou pas).
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 modifiée en accord avec les paramètres définis.

6. Prototypes et choix techniques

6.1 language de programmation

Au vu de l'analyse de l'existant et des contraintes du projet, nous avons choise d'utiliser le langage C++ pour développer nottre application. En effet, une grande partie des projets et des librairies traitant du sujet utilisent le C++, notamment pour des raisons d'efficacité de gestion de la mémoire qui est un point crucial du projet.

De même, l'utilisation de la bibliothèque OpenGL sera grandement facilitée car nous avons déjà tous une certaine expérience à l'utiliser avec ce langage. Pour s'assurer que notre programme puisse fonctionner sur différents systèmes et configurations, nous utilisons OpenGL 3.3.

6.2 Implémentation du module CDLOD

Pour commencer à réaliser la transformation des heightmaps en quadtrees, nous sommes partis de heightmaps simples préconstruites et distribuées sous license libre.

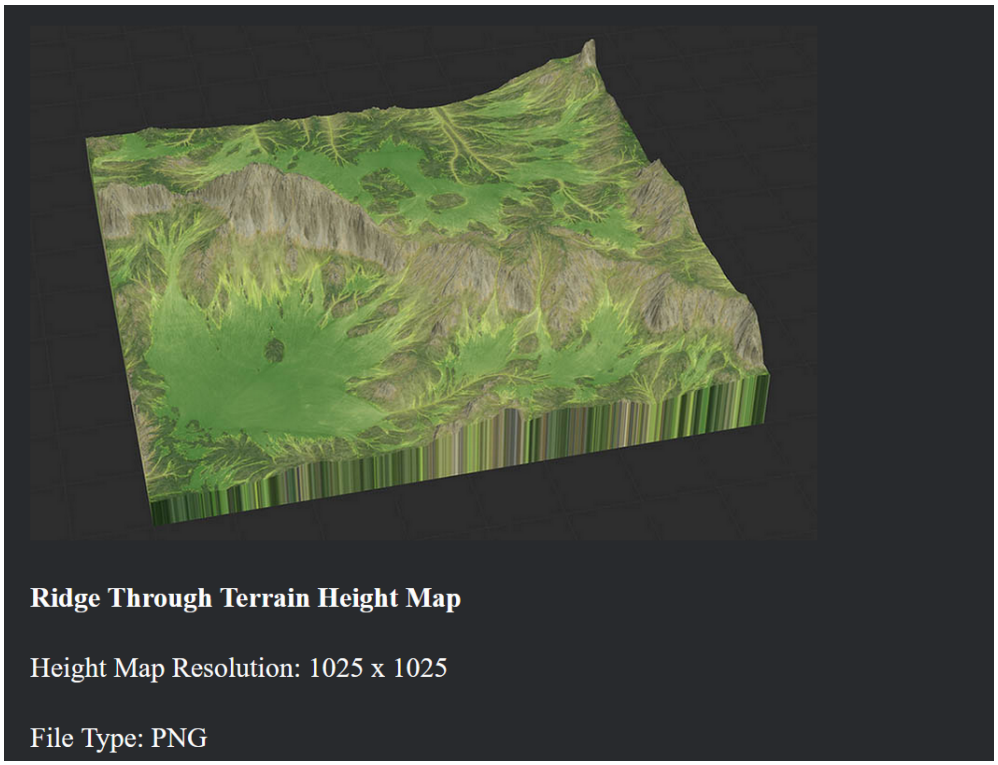


Figure 6.1: Heightmap plane 1025x1025, PNG
https://www.motionforgepictures.com/height-maps

La première contrainte apparaissant ici est celle du format des heightmaps. En effet, les cartes que nous avons trouvé ne sont pas en .RAW comme nous le pensions mais dans différents formats image tels que PNG, TIFF ou encore EXR. Nous avons donc commencé par utiliser la librairie libpng pour traiter cette première heightmap.

7. Architecture de l'application

7.1 Explication du diagramme de classes

Notre classe Application est le coeur du logiciel, c'est elle qui via la méthode setUp va instancier l'input manager, la génération de terrain et le Viewer, c'est donc elle qui va faire le choix de quel algorithme utiliser (CDLOD dans notre cas) et quel rendu 3D (OpenGL dans notre cas).

OpenGLViewer possède une mesh et un shader lui permettant de faire le rendu 3D de la sphère, ainsi que trackball qui lui permet de fixer la caméra sur la sphère, à noter que trackball et caméra sont indépendant d'OpenGLViewer.

L'Application pourra choisir quel algorithme utiliser, pour cela on utilise le pattern "Builder", ainsi le client (ici l'Application) doit choisir l'algorithme puis compiler celui-ci, une fois fait il pourra récupérer le résultat en indiquant le niveau de détail qu'il souhaite (spécifique à CDLOD).

Enfin l'algorithme utilise un MapLoader qui nous permet depuis un fichier PNG / TIFF / EXR ou depuis une génération de HeightMap aléatoire utilisant libnoise d'importer une HeightMap sous un format Unsigned Int sur 8 bits, car les HeightMap utilise des nuances de gris allant de 0 à 255. Celui ci sera transmis à OpenGL via l'application qui devra l'utiliser afin de dessiner les mesh.

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.