REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mohamed Khider – BISKRA
Faculté des Sciences Exactes, des Sciences de la Nature et de la Vie
Département d'informatique

N° d'ordre : IVA 5 /M2/2018

Mémoire

Présenté pour obtenir le diplôme de master académique en

Informatique

Parcours : **Images et vie artificielle**

Rendu des terrains basé sur le Displacement Mapping

Par :
Moussaoui Oussama

Soutenu le 25/06/2018, devant le jury composé de :

Mr.Cherif Foudil	Professeur	Président
Mr.Belaiche Hamza	M A A	Rapporteur
Mr.Boucetta Mebarek	M A A	Examinateur

Dédicaces

A l'homme de ma vie, mon exemple éternel, mon soutien moral et source de joie et de bonheur, celui qui s'est toujours sacrifié pour me voir réussir, que dieu te garde dans son vaste paradis, à toi mon père.

A la lumière de mes jours, la source de mes efforts, la flamme de mon cœur, ma vie et mon bonheur ; maman que j'adore.

Aux personnes dont j'ai bien aimé la présence dans ce jour, à tous mes frères Salah Eddine et Yacine ,et mes sœurs Dr. Hadjer et Fatima Zahra, je dédie ce travail dont le grand plaisir leurs revient en premier lieu pour leurs conseils, aides, et encouragements.

Aux personnes qui m'ont toujours aidé et encouragé, qui étaient toujours à mes côtés, et qui m'ont accompagnaient durant mon chemin d'études supérieures, mes aimables amis, collègues d'étude, et mon frère de cœur, toi Younes Rouabeh.

Remerciements

En préambule à ce mémoire nous remerciant ALLAH qui nous aide et nous donne la patience et le courage durant ces langues années d'étude.
Nous tenant à remercier sincèrement Monsieur, Belaiche Hamza *, qui, en tant que Directeur de mémoire, s'est toujours montré à l'écoute et très disponible tout au long de la réalisation de ce mémoire, ainsi pour l'inspiration, l'aide et le temps qu'il a bien voulu nous consacrer et sans qui ce mémoire n'aurait jamais vu le jour.*

Je remercie également les membres du jury :

Mr.Cherif Foudil	*Professeur*
Mr.Boucetta Mebarek	*M A A*

D'avoir accepté l'évaluation de ce travail.
J'exprime ma profonde reconnaissance à tous ceux qui m'ont aidé à l'élaboration de ce mémoire.

Résumé

Le rendu à grand maillages d'un terrain détaillés est une question importante dans de nombreuses applications graphiques. L'objectif, comme pour de nombreuses algorithmes de rendu en temps réel, est de fournir un niveau de détail élevé (LOD) sans sacrifier l'efficacité et la performance. Les unités de traitement graphique modernes (GPU) présentent un haut degré de parallélisme et, au fil des ans, ont adopté un nombre croissant de techniques pour accélérer le rendu réaliste. Une telle technique est la tessellation, c'est-à-dire la subdivision récursive d'éléments d'objet en des parties plus fines ou plus grossières dans le but d'obtenir la quantité de détails appropriée. Le but de ce projet était d'explorer quelques concepts de base pour utiliser un algorithme de tessellation adaptative avec l'accélération GPU pour rendre de grands terrains de cette manière. Un prototype a été créé en utilisant OpenGL, avec des shaders GLSL (vertex, contrôle de tessellation, évaluation de la tessellation et fragment). Le plus grand avantage de la tessellation est la réduction de la bande passante entre la mémoire et le GPU. Tessellation améliore FPS, car il est plus rapide de contrôler les sommets sur le GPU que sur le CPU.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Le rendu de terrain est un domaine de l'infographie qui couvre les méthodes de visualisation des surfaces imaginaires et réelles en temps réel. Il a de nombreuses applications, y compris les systèmes d'information géographique (SIG) , la simulation de vol, les systèmes de vision synthétique (SVS) et les jeux informatiques. Ces applications demandent une faible latence, tout en nécessitant des images précises et réalistes. Bien que les approches actuelles atteignent un réalisme élevé, en raison de la nature vaste et très détaillée des scènes de terrain, les approches modernes limitent la taille du terrain. Ceci est généralement caché au spectateur en appliquant des effets de brouillard à des zones éloignées, pour le cacher.

L'un des principaux objectifs dans le domaine de l'infographie a toujours été l'effort continu pour visualiser la scène donnée aussi réaliste que possible. Puisque la plupart des scènes modèlent le monde de la vie réelle, elles sont très susceptibles de contenir une sorte de paysage ou de terrain à moins qu'elles ne soient visualisées par ex. intérieurs de construction ou monde au niveau moléculaire.

Cependant, dans le cas où une source de données appropriée pour le terrain n'est pas disponible ou que nous sommes limités par la mémoire de l'ordinateur, c'est l'occasion de générer notre propre surface de terrain à l'aide d'un algorithme de génération de terrain.

Une de nos options est de simuler des lois physiques mais cette technique est trop exigeante en performance et ne fait pas partie de cette thèse.

Par conséquent, nous allons étudier un groupe d'algorithmes qui approximent les surfaces qui ressemblent à un terrain naturel.

La famille d'algorithmes de rendu de terrain la plus courante est appelée terrain mappé en hauteur, qui utilise une image (heightmap) pour stocker les données d'altitude. Les heightmaps sont généralement créés à partir de données satellitaires, connues sous le nom de cartes d'élévation numériques (DEM), ou peuvent être créées par un artiste à l'aide d'un outil de modélisation. Chaque pixel représente l'altitude d'un point sur la surface, qui est utilisé par l'algorithme de rendu du terrain pour produire une représentation 3D du terrain. Les pixels sont généralement 16 bits et varient. La technique la plus courante pour y parvenir est de traiter le terrain comme une grille 3D, où l'élévation de chaque point est décalée en fonction de l'altitude.

Certaines des techniques proposées pour le rendu du terrain ne sont pas limitées à la visualisation du terrain. Les techniques ont été appliquées à d'autres domaines, tels que le rendu de l'océan. Les techniques sont les plus bénéfiques pour rendre les modèles qui ne peuvent pas

facilement tirer parti des techniques d'optimisation traditionnelles comme, culling.

La recherche de rendu de terrain, comme de nombreux domaines de l'infographie, est motivée par le désir de produire des scènes 3D de haute qualité le plus rapidement possible. La demande des consommateurs est l'un des facteurs qui expliquent cela en raison des attentes et des demandes croissantes. À mesure que le matériel devient plus rapide et que les fonctionnalités de l'API sont introduites, de nouvelles techniques deviennent possibles. La recherche sur le rendu du terrain étudie comment ces facteurs peuvent être appliqués pour produire des terrains plus grands et plus réalistes en temps réel.

Objectif : Le but de notre travail concerne d'étudier les améliorations possibles des algorithmes de rendu de terrain existants, en se concentrant sur l'utilisation des caractéristiques matérielles et API récentes. Nous étions particulièrement intéressés à savoir si les algorithmes plus anciens pouvaient bénéficier de l'une des fonctionnalités API récemment introduites, ainsi que des fonctionnalités matérielles récentes. Nous voulions voir si nous pouvions améliorer les performances et réduire la complexité de la mise en œuvre, tout en conservant une qualité d'image élevée.

Axes du mémoire : Pour assurer les objectifs de notre travail, nous organisons ce mémoire en quatre chapitres :

- **Chapitre 1 " Aperçu générale de l'infographie : "** Ce chapitre donne un aperçu générale de l'infographie, en mettant l'accent sur les idées et les concepts que nous avons utilisés.
- **Chapitre 2 " Génération et texturisation de terrain : "** Exposer les méthodes et les algorithmes de rendu du terrain précédents et les catégorise en fonction des caractéristiques. D'un autre côté ce chapitre présente un ensemble de techniques d'habillage de terrain, qui consiste à colorer le terrain en espérant obtenir un résultat suffisamment réaliste.
- **Chapitre 3 " Implémentation et Résultats : "**Contient les résultats des expériences qui ont été réalisées pour évaluer notre travail et nous terminerons notre mémoire par une conclusion.

Chapitre 1

Génération et texturisation de terrain

1.1 Introduction

Depuis le milieu du XXe siècle, diverses techniques de représentation numérique du terrain ont été développées avec le développement de la technologie informatique, des mathématiques modernes et de l'infographie. De nos jours, l'utilisation de l'ordinateur est devenue un repère important dans l'ère de l'information. En effet, les ordinateurs sont devenus un moyen important pour la représentation de la surface du terrain numérique. Les surfaces de terrain numériques peuvent être représentées mathématiquement et graphiquement. Les séries de Fourier et les polynômes sont des représentations mathématiques courantes. Grille régulière, grille irrégulière, contournage et le diagramme en coupe sont des représentations graphiques courantes.

Ce chapitre explique les méthodes de génération de techniques procédurales pour créer des paysages aléatoires, y compris les algorithmes de bruit, les algorithmes d'érosion, la modélisation de l'eau et les techniques de simulation de végétation.

D'un autre côté ce chapitre étudie les recherches relatives de génération de textures de terrain en temps réel. Ces textures de surface sont définies comme une texture représente la couleur de surface à tout point donné du maillage. Ces textures sont composées de différents matériaux (tels que gazon, roches, neige, etc.) afin de reproduire de manière authentique la diversité et les détails sans fin du monde réel.

1.2 Catégorisation

Les méthodes de génération de terrain peuvent être divisées en trois catégories principales en fonction de leur approche du problème :

- **Modélisation basée sur la physique :** Ces méthodes simulent des processus du monde réel en appliquant des lois physiques sur des objets virtuels. Cela conduit à des résultats très réalistes, qu'ils soient statiques ou dynamiques. D'un autre côté, ces méthodes sont très exigeantes en termes de performance et impliquent généralement beaucoup d'ité-

rations avec des milliers à des millions de petites particules. Pour cette raison, ils sont principalement utilisés uniquement dans la recherche et les applications scientifiques.

- **Algorithmes d'approximation :** Ces méthodes ne simulent aucune loi physique mais essaient seulement d'approximer le résultat final des processus de mots réels. Ils visent généralement une approximation visuellement suffisante tout en gardant le coût de calcul final aussi bas que possible. La base théorique de ces méthodes réside habituellement dans la théorie fractale ou la synthèse du bruit.
- **les méthodes de synthèse à partir d'exemples :** Ces méthodes ne permettent de reproduire qu'un seul type de relief homogène aux images données. Il n'existe pas, à ce jour, d'algorithme permettant une modification interactive de terrains complexes avec un haut niveau de détails au sol, et permettant un placement rapide et intuitif d'éléments caractéristiques géologiques et géographiques en utilisant des opérations d'édition simples. Un problème connexe vient du fait que beaucoup de ces algorithmes ne permettent pas de représenter les terrains à différentes échelles : les terrains sont modélisés sur des structures de grilles qui représentent des reliefs à une précision fixe. Ces terrains sont souvent stockés sous la forme de cartes de hauteurs qui peuvent être converties, plus tard, en des maillages adaptés à une visualisation rapide avec des mécanismes de niveaux de détails.

1.3 Fractales

Puisque de nombreux algorithmes de génération de terrain sont basés sur l'approximation de fractales, cette section introduit deux concepts importants de la théorie fractale qui a été conçue et introduite par Benoît B. Mandelbrot au cours des années 1967-1982 [1].

1.3.1 Principe et théorie

Une approche intuitive consiste à considérer qu'une fractale est la transformation récursive d'un objet par n copies de lui même à l'échelle r (avec r < 1) [2].

— Cette propriété est appelée l'auto-similarité.
— On a coutume de caractériser un objet fractal à partir d'un paramètre numérique généralement noté D appelé dimension fractale.
— D est égal à $\frac{log(n)}{log(\frac{1}{r})}$. Il caractérise la manière selon laquelle une fractale évolue dans l'espace où elle est dessinée. Plus D est grand plus l'évolution est "chaotique".

1.3.2 Auto-similarité

Un objet ou sa partie est appelé auto-similaire si «chacune de leur portion peut - dans un sens statistique - être considérée comme une image à échelle réduite de l'ensemble». [3].
L'un des exemples les plus simples est la ligne dans l'espace 2D - s'il n'y a pas d'autre objet

auquel nous pouvons comparer la ligne, il n'y a aucun moyen de découvrir quelle partie de la ligne est affichée.
Parmi les exemples plus compliqués de formes planaires auto-similaires, on trouve par exemple. Triangle de Sierpinski ou flocon de Koch, vu sur figure 1.1.

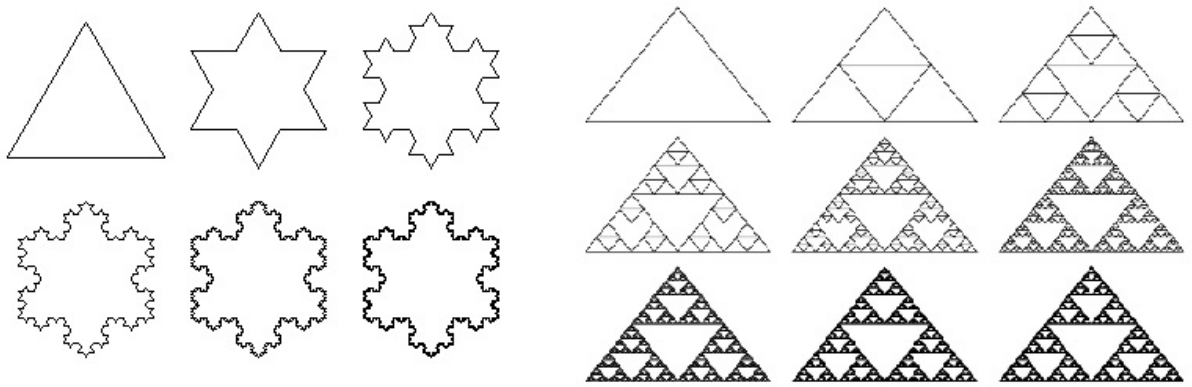


FIGURE 1.1 – Flocon de neige de Koch (à gauche) et Triangle de Sierpinski(à droite).

1.3.3 Dimension fractale

La dimension fractale est une extension d'une dimension spatiale de base qui permet d'utiliser un nombre réel non-négatif $D \in R$. Pour une ligne courbe située dans le plan 2D $D \in \langle 1,2 \rangle$. De même, pour des surfaces courbes dans l'espace 3D $D \in \langle 2,3 \rangle$ [3].

1.4 Valeurs stochastiques

La plupart des fractals biens connues telles que le flocon de neige de Koch, l'éponge de Menger et d'autres sont trop régulières et leur apparence est très artificielle. Afin d'obtenir une courbe ou une surface d'apparence naturelle, il est nécessaire de randomiser le processus de création de surface fractale d'une manière ou d'une autre. Dans cette section, les informations de base relatives aux nombres aléatoires sont expliquées avec deux fonctions de distribution de probabilité qui sont utilisées plus loin dans le texte dans plusieurs algorithmes. Pour plus d'informations sur le sujet.

1.4.1 Générateur de nombres aléatoire ou pseudo-aléatoire

Le générateur de nombres pseudo-aléatoires est une formule logicielle spéciale permettant de générer une suite de nombres qui semblent aléatoires. Cependant, la séquence est définie par un paramètre de générateur initial (ou ensemble de paramètres) appelé *speed* et puisque le calcul est déterministe, nous obtenons toujours la même séquence pseudo-aléatoire avec la même graine.

En raison de leur relative simplicité, les générateurs de nombres pseudo-aléatoires sont principalement utilisés dans des applications simples. La majorité des langages de programmation communs contiennent des fonctions spéciales ou des bibliothèques qui permettent au programmeur de travailler facilement avec des nombres pseudo-aléatoires (par exemple les fonctions *srand()* et *rand()* en C / C ++). Cependant, sa nature déterministe les disqualifie de l'utilisation dans des domaines avancés comme la cryptographie.

Si nous souhaitons obtenir une séquence de nombres réellement aléatoires basée sur un processus non-déterministe, nous pouvons utiliser un générateur de nombres aléatoires. C'est un dispositif matériel spécial qui génère une séquence de nombres aléatoires basés sur des processus physiques tels que le bruit thermique, l'effet photoélectrique ou d'autres qui sont en théorie complètement imprévisibles. Comme ces périphériques matériels ne sont généralement pas largement disponibles, il est bien sûr possible de partager leurs résultats sur le réseau localement ou globalement.

Puisque les générateurs de nombres pseudo-aléatoires offrent des approximations suffisamment bonnes, les vrais générateurs de nombres aléatoires ne sont pas nécessairement requis dans le domaine de la génération de terrain. La seule exception est la modélisation basée sur la physique où la simulation et les vraies variables aléatoires jouent souvent un rôle important.

1.4.2 Distribution uniforme

La distribution uniforme est l'une des distributions de probabilité de base qui a à la fois une forme discrète et continue.

- **La distribution uniforme discrète** représente un cas où l'on choisit un objet aléatoire parmi un nombre fini d'objets $n \in \mathbf{N}$ et où chaque objet a une probabilité égale $\frac{1}{n}$ à sélectionner.
- **La distribution uniforme continue** est définie par sa fonction de densité de probabilité qui est représentée dans l'équation 1.1, où a et b sont les limites de l'intervalle pour lequel la probabilité est calculée. Puisque la distribution est uniforme, tous les intervalles également longs ont la même probabilité.

$$U(a,b) = \begin{cases} \frac{1}{b-a} & Si \qquad x \in \langle a,b \rangle \\ 0 \end{cases} \tag{1.1}$$

1.4.3 Distribution normale

La distribution normale (ou gaussienne) est l'une des distributions de probabilité continues les plus connues et les plus utilisées.Sa fonction de densité de probabilité est représentée dans l'équation 1.2 où μ est la valeur moyenne et σ^2 est la variance. La distribution normale avec $\mu = 0$ et $\sigma = 1$ est appelée distribution normale standard et le graphique de sa fonction de densité de probabilité est affiché dans figure 1.2.

$$N(\mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}}.e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{1.2}$$

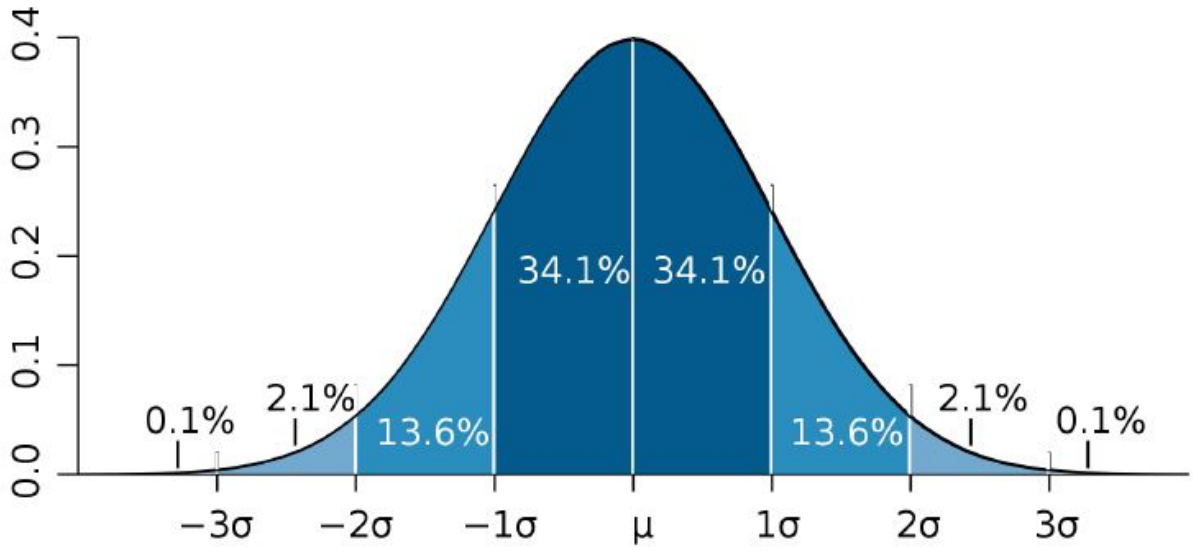


FIGURE 1.2 – Distribution normale standard $N(0,1)$.

1.5 Algorithmes de génération de terrain

1.5.1 Déplacement médian

L'algorithme du déplacement médian (aussi appelé diamant carré, nuage ou plasma) est un algorithme de génération de terrain introduit en 1982 par Fournier, Fussell et Carpenter avec comme objectif principal une approximation plus facile et plus rapide du mouvement brownien fractionnaire bidimensionnel [4].

1.5.1.1 Déplacement médian en 1D

La version unidimensionnelle de l'algorithme fonctionne de la façon suivante, comme décrit par le papier original en 1982 :
Soit $I = \langle f_1, f_2 \rangle$ être un intervalle discret avec régulièrement positionné points.

1. Définissez les valeurs de coin f_1 et f_2 sur les valeurs générées par la distribution normale.
2. Calculer la valeur de la hauteur du point central.

$$f_{mid} = \frac{f_1 + f_2}{2} + \varepsilon \tag{1.3}$$

   où ε est une erreur pseudo-aléatoire (décrite en détail dans la section 1.5.1.3).

3. Répétez récursivement les étapes 2.-3. aux intervalles $\langle f_1, f_{mid} \rangle$ et $\langle f_{mid}, f_2 \rangle$ jusqu'à ce qu'ils contiennent plus d'un point (c'est-à-dire que f_{mid} est distinct des points d'angle f_1 et f_2).

Cette version de l'algorithme est limitée par un nombre donné de points et donc par un niveau de détail. Néanmoins, il est possible de l'implémenter d'une manière où d'autres itérations sont également calculées en zoomant sur la courbe.

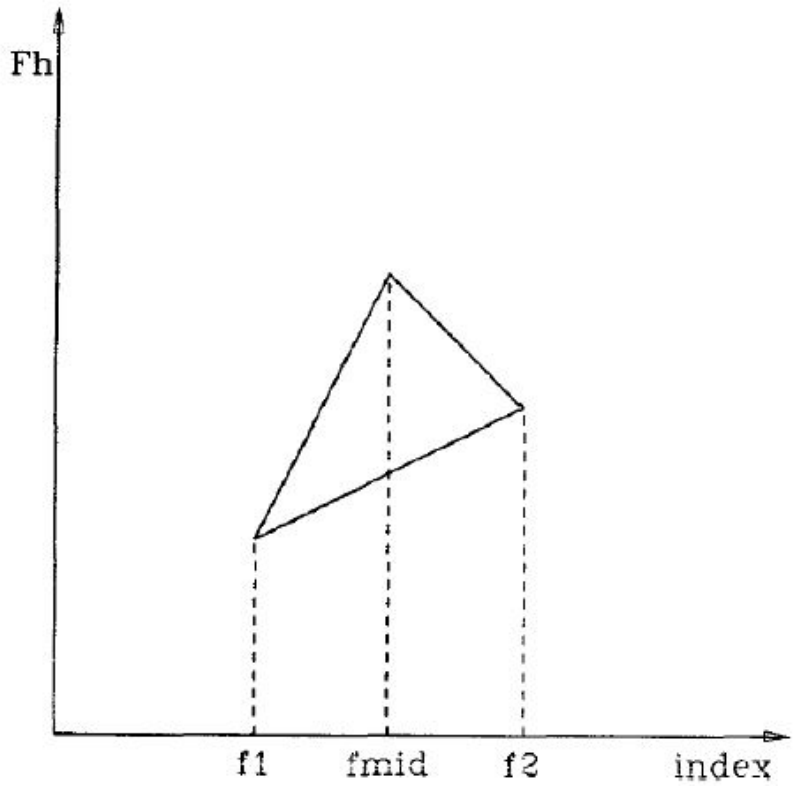


FIGURE 1.3 – Déplacement du milieu en 1D.

1.5.1.2 Déplacement médian en 2D

La version bidimensionnelle de l'algorithme fonctionne de la manière suivante, comme décrit par l'article original en 1982 :
Soit G un champ de hauteur régulier avec les coins A, B, C et D.

1. Définissez les valeurs de coin de A, B, C et D sur les valeurs prédéfinies ou pseudo-aléatoires générées par la distribution normale (voir figure 1.4, partie a).
2. Calculez la valeur de hauteur du point central (voir figure 1.4, partie b).

$$S = \frac{A+B+C+D}{4} + \varepsilon \tag{1.4}$$

   où ε est une erreur pseudo-aléatoire (décrite en détail dans la section 1.5.1.3).

3. Calculez les valeurs de hauteur des points au milieu des quatre côtés en tant que moyenne des coins adjacents (voir figure 1.4, partie c).
   $U = \frac{A+B}{2}$,$V = \frac{C+D}{2}$,$X = \frac{A+C}{2}$ and $Y = \frac{B+D}{2}$
4. Répétez récursivement les étapes 2.-4. sur quatre sous-rectangles donnés par les coins (A,U,X,S), (U,B,S,Y), (X,S,C,V) et (S,Y,V,D) jusqu'à ce qu'ils contiennent plus de quatre points d'angle (voir figure 1.4, parties d et e).

Tout comme dans le cas unidimensionnel, cette version de l'algorithme est limitée par un nombre donné de points et donc par un niveau de détail.Néanmoins, il est possible de l'implémenter d'une manière où d'autres itérations sont également calculées en zoomant sur la surface.

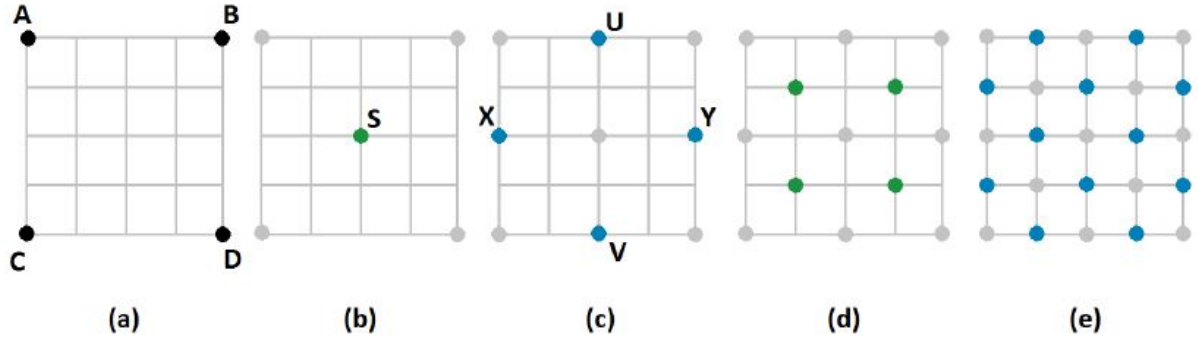


FIGURE 1.4 – Exemple d'algorithme de déplacement médian (diamond-square) sur une grille carrée de 5x5 points.

1.5.1.3 Valeur d'erreur pseudo-aléatoire

L'erreur pseudo-aléatoire qui est ajoutée à la valeur centrale forme le noyau de l'algorithme. Si elle était omise, l'algorithme produirait une surface plate inclinée (ou ligne, dans un cas unidimensionnel) - l'apparence et la qualité globales du terrain est affecté uniquement par cette valeur.
Plusieurs formules de calcul de la valeur ont été proposées et utilisées depuis l'introduction de l'algorithme.

L'article original proposait les formules 1.5 et 1.6 pour la version unidimensionnelle et bidimensionnelle de l'algorithme, respectivement [4].

$$\varepsilon = N(0, 2^{-H}.scale^{level}) \tag{1.5}$$

$$\varepsilon = N(0, c^{-H.level}) \tag{1.6}$$

La valeur d'erreur pseudo-aléatoire est dans les deux cas générée comme une valeur gaussienne avec une valeur moyenne nulle. Le paramètre H représente l'exposant de Hurst tandis que *level* de paramètre signifie la profondeur de récursion, augmentant à chaque itération. *Scale* peut être utilisée pour changer l'échelle de toute la courbe unidimensionnelle et c est une *«constante à ajuster pour s'adapter à l'application»* [4, p.381].

Même si les formules proposées donnent de très bons résultats si elles sont correctement définies, elles contiennent beaucoup de variables et sont un peu difficiles à utiliser. Par conséquent, d'autres formules ont été conçues pour le simplifier.

$$\varepsilon = N(0, \frac{1}{2^{2H(level+1)}}\sigma^2) \tag{1.7}$$

L'un d'eux est par exemple formule 1.7, qui peut être utilisée à la fois dans un cas bidimensionnel et bidimensionnel.Hurst exposant et le niveau de récursion sont partagés avec la formule précédente, le seul nouveau paramètre est la variance initiale σ^2. Cette formule est utilisée plus tard même dans l'algorithme des fautes aléatoires (voir les équations 1.10 et 1.11).

$$\varepsilon = (1-H).N(0,1).|A,B| \tag{1.8}$$

Pour simplifier encore plus, il est possible d'utiliser la formule 1.8, qui est basée sur la distance des points A et B dans les deux dimensions. Plus les points sont proches, plus la variance de la distribution normale est petite.

Chaque formule génère une surface ressemblant plus ou moins au terrain réel - mais le choix de celui à utiliser dépend uniquement de l'utilisateur, de son application et de ses besoins.

1.5.1.4 La complexité temporelle

Dans le cas d'une implémentation récursive, la hauteur de chaque point de surface est évaluée exactement une fois. Même si le nombre de points voisins nécessaires pour le calcul varie pour différents points, il n'y a jamais plus de 4 points voisins. La mise en œuvre récursive de l'algorithme de déplacement du point médian (sur un seul processeur) pour une surface donnée constituée de $n \in \mathbf{N}$ points a donc une complexité temporelle linéaire *O(5n)*, c'est-à-dire *O(n)*.

1.5.2 Défauts aléatoires

Les fautes aléatoires sont l'une des méthodes de génération de terrain les plus simples, dont la première version a été introduite en 1994 par Robert Krten [5]. Cette méthode vise à simuler des forces de la nature telles que la séparation des plaques tectoniques et l'érosion des rives[6, p.499].

1.5.2.1 Principe de base

L'idée principale de l'algorithme est basée sur une corruption pseudo-aléatoire d'un maillage polygonal qui est interprété comme une heightmap.

C'est une méthode itérative qui commence par une grille rectangulaire bidimensionnelle discrète Ω et une valeur de défaut δ.

Dans son étape initiale, toutes les valeurs de hauteur de Ω sont définies à la même hauteur initiale avec la valeur de défaut δ qui est réglé à sa valeur initiale. Ensuite, les étapes suivantes sont répétées $n \in \mathbf{N}$ fois :

1. Deux points A et B où $A \neq B$ et qui se trouvent tous deux dans la grille sont choisis de manière pseudo-aléatoire.
2. Les points A et B déterminent clairement la ligne l qui contient les deux points.
3. Les valeurs de hauteur de tous les points de la grille Ω qui se trouvent d'un côté de la ligne l sont modifiées par $+\delta$. La hauteur de tous les autres points est modifiée par $-\delta$.
4. Diminuer la valeur de défaut δ par une formule prédéfinie $-$ mais de telle sorte qu'elle reste supérieure à 0.

L'algorithme est généralement automatiquement arrêté après certains prédéfinis nombre d'itérations n. Bien sûr, il peut également être implémenté de manière à calculer les prochaines itérations jusqu'à ce que le résultat soit visuellement approprié et qu'il doive donc être arrêté manuellement.

Nous pouvons voir le maillage polygonal corrompu dans trois différentes étapes de calcul sur figure 1.5. Sur l'image de gauche, elle est affichée après la première itération où la surface a été divisée en deux moitiés. Sur l'image centrale, elle est affichée après la quatrième itération

dans laquelle la valeur de défaut a déjà une valeur plus petite. Enfin, à droite, l'image montre le résultat après l'itération finale.

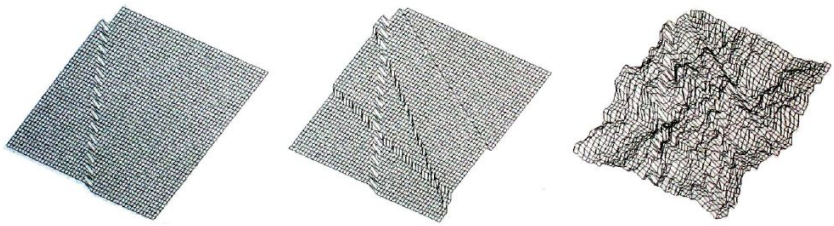


FIGURE 1.5 – Plusieurs itérations sélectionnées de l'algorithme des fautes aléatoires.

1.5.2.2 Diminuer la valeur de défaut

En utilisant une formule appropriée pour diminuer la valeur de défaut δ est crucial si nous souhaitons obtenir des résultats naturels.

La formule utilisée dans l'algorithme original est montrée en 1.9 où $n \in \mathbf{N}$ est le nombre d'itérations et $i \in \mathbf{N}$ est l'itération courante [5].

$$\delta_i = \frac{n}{i} \tag{1.9}$$

Une autre approche pour diminuer la valeur de défaut est basée sur la distribution normale [24, p. 281]. Ici, ce n'est pas la valeur de défaut δ que nous diminuons continuellement mais la variance σ_i^2 d'une distribution gaussienne $\mathcal{N}$. Cette distribution avec une valeur moyenne de zéro est alors utilisée pour générer pseudo-aléatoire le nouveau δ

L'entrée initiale est constituée de l'exposant de Hurst H et de la variance initiale σ^2. Ces valeurs sont dans chaque étape utilisée dans la formule 1.10 pour calculer la variance diminuée σ_i^2.

$$\sigma_i^2 = \frac{1}{2^{2H(i+1)}}\sigma^2 \tag{1.10}$$

La variance calculée est ensuite utilisée pour générer de façon pseudo-aléatoire une nouvelle valeur de défaut δ par l'équation 1.11.

$$\delta = \mathcal{N}(0, \sigma_i^2) \tag{1.11}$$

Le plus grand avantage mais aussi un inconvénient de cette approche est son adaptabilité par de nombreux paramètres (exposant de Hurst, variance initiale, nombre d'itérations, etc.) - même s'il est possible de trouver des paramètres qui donneront un terrain visuellement beau, beaucoup des combinaisons produisent des résultats d'apparence anormale.

Par conséquent, la δ la valeur n'est habituellement réduite linéairement que par une simple équation 1.12, ou par une modification de celle-ci. Cette approche donne un terrain visuellement très attrayant.

$$\delta_i = \delta_0 + (i/n)(\delta_n - \delta_0) \tag{1.12}$$

1.5.2.3 Lissage(Smoothing)

La surface du terrain générée par l'algorithme des fautes aléatoires est susceptible d'être très rugueuse et tranchante avec des changements d'élévation soudains et anormaux. Comme indiqué par [5], *"ce terrain est utilisable pour certaines applications sans autre modification, mais pourrait être rendu plus réaliste en lissant les aspérités"*.

Pour lisser le terrain, nous pourrions utiliser certaines des techniques de traitement d'image numérique telles que les filtres passe-bas ou mieux, une érosion de la morphologie mathématique.

1.5.2.4 Modifications

Il est possible de modifier la version de base de l'algorithme de plusieurs façons :

- **Forme heightmap** - Au lieu d'utiliser une grille rectangulaire comme une heightmap, il est possible d'utiliser toute autre forme plane comme base de terrain. De plus, l'algorithme des défauts aléatoires est également approprié pour la génération spatiale de par ex. surface des planètes entières comme décrit dans [7].

- **Ligne de division** - D'autres formes planes (par exemple une ellipse, un rectangle ou un polygone convexe ou concave) peuvent également être utilisées pour diviser le plan de la hauteur en deux segments. Cependant, nous devons empêcher une augmentation ou une diminution continue des valeurs de hauteur du terrain, de sorte que l'addition et la soustraction des segments soient échangées de façon répétée.

- **Itération initiale et valeur de défaut** - Dans certains cas, l'algorithme peut également être utilisé sur un maillage polygonal qui n'est pas entièrement plat mais contient déjà des données de terrain. Plusieurs itérations d'algorithme de fautes aléatoires avec une faible valeur de défaut (de préférence constante) conduiront donc à une déformation et à une distorsion du terrain d'origine, ce qui peut parfois être exactement ce que l'on souhaite réaliser.

- **Fonction d'échelon** - Une augmentation et une diminution de deux moitiés de la surface du terrain équivaut à appliquer une fonction d'échelon basée sur la distance du point à la ligne [8].

On peut remplacer la fonction pas par une fonction différente, de préférence lisse, dont on peut voir des exemples sur figure 1.6.

En cas d'utilisation par ex. onde sinusoïdale ou cosinusoïdale, le résultat final sera également beaucoup plus lisse et ne nécessitera pas de filtrage supplémentaire comme dans le cas de la fonction de base (comme le montre figure 1.7).

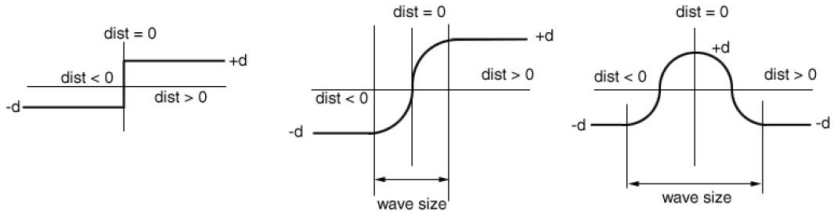


FIGURE 1.6 – Diverses fonctions de défaut - Fonction de pas de base (gauche), onde sinus (centre) et onde cosinus (droite).

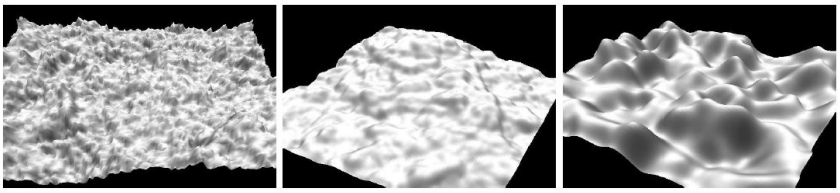


FIGURE 1.7 – Résultats après itération finale en utilisant différentes fonctions de défaut - Fonction de pas de base (gauche), onde sinus (centre) et onde cosinus (droite).

1.5.2.5 La complexité temporelle

- Puisque l'évaluation de chaque point a une complexité temporelle constante *O(1)*, le calcul de $n \in \mathbf{N}$ points a une complexité temporelle linéaire *O(n)*.
- En cas d'itérations multiples $k \in \mathbf{N}$, la complexité globale du temps est *O(k.N)*. Cependant, puisque k est une constante, la complexité asymptotique est toujours linéaire, c'est-à-dire *O(n)*.

1.5.3 Bruit de Perlin

1.5.3.1 Principe de base

Le bruit de Perlin est une fonction de bruit cohérente conçue en 1982 par Ken Perlin. La fonction de bruit elle-même est une application de l'espace à n dimensions à la valeur réelle unique 1.13. Cependant, habituellement seulement $n \in \langle 1,4 \rangle$ est utilisé [9, p.14].

$$\mathbf{R}^2 \longmapsto \mathbf{R}, n \in \mathbf{N} \tag{1.13}$$

1.5.3.2 Bruit cohérent ou non cohérent

La cohérence est l'une des propriétés les plus importantes du bruit de Perlin. C'est la propriété clé qui donne à la texture obtenue un aspect lisse, sans transitions nettes. Trois conditions doivent s'appliquer pour que le bruit soit marqué comme cohérent [10] :

1. Passer dans la même valeur d'entrée retournera toujours la même valeur de sortie.
2. Une petite modification de la valeur d'entrée produira une petite modification de la valeur de sortie.
3. Une modification importante de la valeur d'entrée produira un changement aléatoire de la valeur de sortie.

La différence entre le signal de bruit cohérent et non cohérent en 2D peut être vue sur figure 1.8.

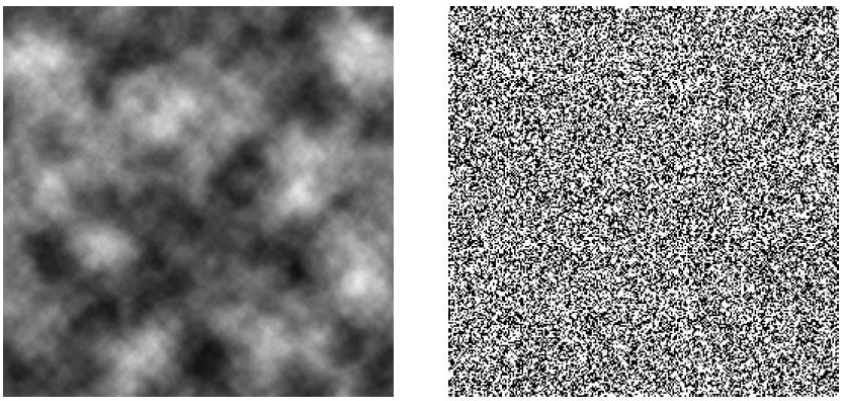


FIGURE 1.8 – Texture de bruit 2D cohérente (gauche) vs. non cohérente (droite).

1.5.3.3 Tableau des dégradés et des permutations

Pour créer l'illusion d'un bruit non-répétitif sans sacrifier la vitesse, Ken Perlin a eu une idée de table de gradient prégénérée [9, p.17].

Dans un cas bidimensionnel, $n \in \mathbf{N}$ vecteurs de gradient pseudo-aléatoires de la forme (x,y) sont précalculés et stockés dans une table de gradient, où n est la taille de pavage de la texture de bruit. Dans son implémentation d'origine, Perlin utilisait $n = 256$, mais il est possible de le remplacer par la taille du carrelage du choix du programmeur en fonction du besoin d'application.

Un tableau de permutation de nombres de 0 à $n-1$ est également créé, mélangé et stocké avec la table de dégradé.

1.5.3.4 Fonction de bruit Perlin en 2D

La fonction de bruit est définie pour tout point réel $P = (x,y) \in \mathbf{R}^2$ de la manière suivante, telle que définie par Perlin dans [9, p.15] :

1. Laisser les points $Q = (x_0, y_0) \cup (x_0, y_1) \cup (x_1, y_0) \cup (x_1, y_1) \in \mathbf{Z}$ être les points les plus proches avec les coordonnées entières.
2. Pour chacun des quatre points Q, obtenez un gradient pseudo-aléatoire G à partir d'une table de gradient (voir 1.5.3.3). Le gradient pour le point donné est obtenu par une fonction $G_t[P_t[P_t[x] + y]]$ où G_t et P_t sont respectivement des tables de gradient et de permutation pré-calculées.
3. Pour chacun des quatre points Q, calculer l'influence du point de coin approprié en calculant la valeur $G.(P - Q)$.
4. Interpolez les deux valeurs d'influence supérieure et inférieure en utilisant une courbe d'accélération dans chaque étape (par exemple $3t^2 - 2t^3$).
5. Interpolez les deux dernières valeurs en utilisant la même courbe d'accélération et renvoyez-la à la suite d'une fonction de bruit Perlin au point (x, y).

1.5.3.5 Composition de plusieurs textures de bruit Perlin

Même si le bruit de base de Perlin a un certain nombre d'utilisations différentes (par exemple, il donne aux objets un aspect grossier et sale), sa véritable application est ailleurs.

De nombreuses façons de combiner des textures de bruit pour obtenir divers effets visuellement attrayants ont été découvertes au cours des années.

Une des approches les plus courantes consiste à calculer une somme pondérée de textures multiples générées par la fonction de bruit Perlin (1.14) avec chacune de ces textures générées à des fréquences différentes [9, p.21].

$$CPNoise2D(x, y) = \sum_{i=1}^{n} \frac{1}{2^i} noise(2^i.x, 2^i.y) \tag{1.14}$$

Les textures de bruit additionnées sont appelées *octaves*. Chaque octave supplémentaire est calculée à la fréquence doublée mais son amplitude est réduite de moitié. Il en résulte une surface avec un bruit de basse fréquence provenant des octaves inférieures créant des montagnes et un bruit de fréquence élevé provenant des octaves supérieures créant des détails supplémentaires. De plus, comme la surface finale est créée à partir de la même texture de bruit initiale (seulement redimensionnée), la texture composée finale est auto-similaire à un certain niveau.

La formule 1.14 stipule que chaque octave supplémentaire est ajoutée à la somme finale avec une amplitude réduite de moitié. Cependant, il peut être approprié de le modifier et de le généraliser afin de permettre un rapport différent entre l'octave la plus grande et la plus petite.

$$CPNoise2D(x, y) = \sum_{i=1}^{n} p^i.noise(2^i.x, 2^i.y) \tag{1.15}$$

Le remplacement de $\frac{1}{2^i}$ par p^i est l'un des moyens d'obtenir l'effet désiré, comme on peut le voir dans la formule 1.15 . Le paramètre p est appelé *persistance* et la plupart du temps $p \in \langle 0,1 \rangle$. La valeur $p = 0,5$ donne la formule précédente, 1.14.

1.5.3.6 La complexité temporelle

Le calcul d'une fonction de bruit Perlin pour un point donné a une complexité temporelle constante *O(1)*. Par conséquent, la complexité temporelle du calcul de la fonction de bruit Perlin pour un ensemble de points $n \in \mathbf{N}$ est linéaire, c'est-à-dire *O(n)*.

Dans le cas de $k \in \mathbf{N}$ textures de bruit Perlin multiples qui sont additionnées, la complexité temporelle est *O(k.N)*. Cependant, puisque k est constant, la complexité globale est toujours *O(n)*.

1.6 Le niveau de détail (LOD)

Le niveau de détail (LOD) [11] est une technique qui consiste à réduire la complexité du modèle 3D selon certaines conditions. Le niveau de détail dépendant de la vue (LOD) [12] est une technique permettant de modifier les détails nécessaires pour représenter un modèle 3D, en fonction de l'emplacement du modèle, par rapport à la caméra. Cela permet d'éviter certaines opérations redondantes en rendant les modèles distants dans les moindres détails, sans impact sur la qualité globale.

1.6.1 Techniques multi-résolution pour le terrain

1.6.1.1 Ascendante et descendante

L'un des principaux facteurs de différenciation des algorithmes LOD de terrain, comme pour les techniques de LOD plus générales, est de savoir s'ils sont descendants ou ascendants dans leur approche du problème de simplification. Dans un algorithme Descendante, nous commençons normalement avec deux ou quatre triangles pour la région entière, puis nous ajoutons progressivement de nouveaux triangles jusqu'à ce que la résolution désirée soit atteinte. Ces techniques sont également appelées méthodes de subdivision ou de raffinement. En revanche, un algorithme ascendante commence par le maillage de plus haute résolution et supprime itérativement les sommets de la triangulation jusqu'à ce que le niveau de simplification souhaité soit atteint. Ces techniques peuvent également être appelées méthodes de décimation ou de simplification [13]. La figure 1.9 illustre ces deux approches de la simplification du terrain.

1.6.1.2 Regular Grids and TINs

Une autre distinction importante entre les algorithmes LOD de terrain est la structure utilisée pour représenter le terrain. Deux approches majeures à cet égard sont l'utilisation de champs de

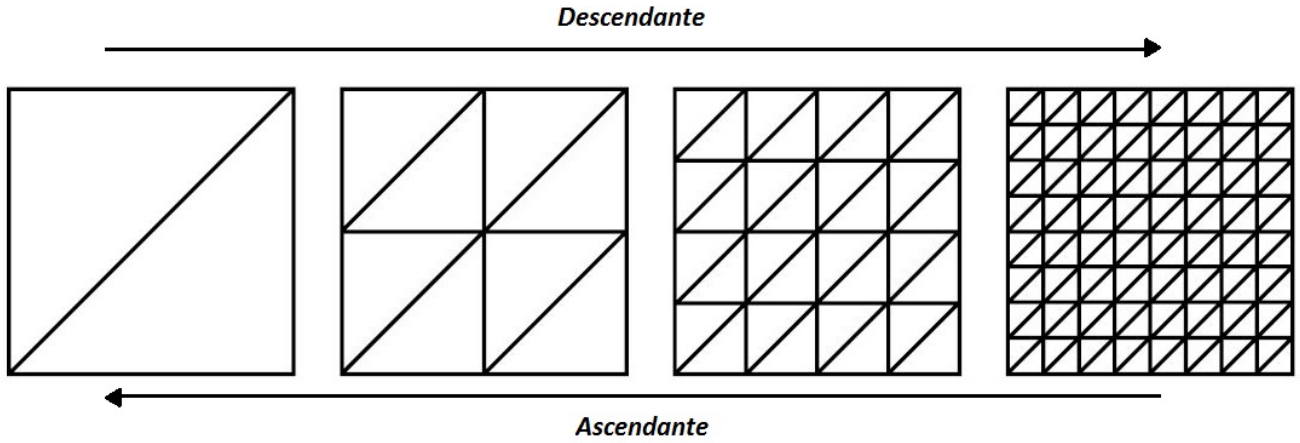
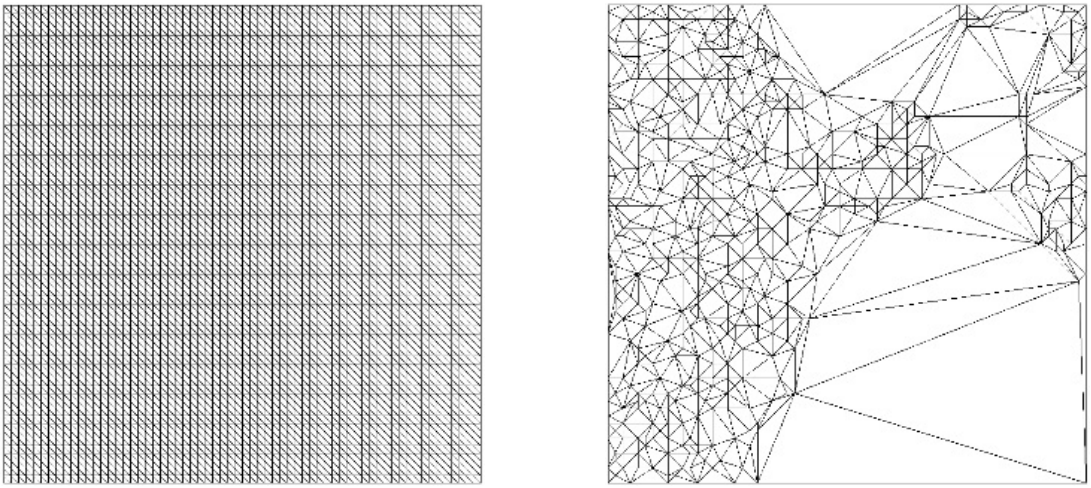


FIGURE 1.9 – Approche descendante et ascendante.

hauteur quadrillés réguliers et de réseaux triangulaires irréguliers (TINs :Triangulated Irregular Networks).Les grilles régulières utilisent une matrice de valeurs de hauteur à des coordonnées x et y régulièrement espacées, tandis que les TIN permettent un espacement variable entre les sommets [14]. La figure 1.10 illustre ces deux approches, montrant une grille régulière de 65 * 65 (égale à 4225) valeurs de hauteur et une représentation de TIN à 512 sommets avec la même précision.



(a) (b))

FIGURE 1.10 – (a) Une représentation régulière du terrain sur la grille, et (b) une représentation du TIN.

1.6.1.3 Quadtrees et Bintrees

Pour implémenter LOD dépendant de la vue pour une structure de grille régulière, nous devons être en mesure de représenter différentes parties de la grille à différentes résolutions. Cela implique une représentation hiérarchique dans laquelle nous pouvons progressivement affiner davantage de détails aux différentes parties de la grille. Il y a un certain nombre d'options

disponibles pour réaliser cette représentation multi-résolution. Les deux plus communs sont le quadtree [15] et l'arbre binaire triangle [13].

- **Une structure de quadtree** est une région rectangulaire est divisée uniformément en quatre quadrants. Chacun de ces quadrants peut ensuite être divisé successivement en quatre régions plus petites, et ainsi de suite (voir figure 1.11(a-d)).
- **Une structure d'arbre binaire en triangle** fonctionne de la même manière qu'un quadtree, mais au lieu de segmenter un rectangle en quatre, il segmente un triangle en deux moitiés. Le triangle racine est normalement défini comme étant un triangle rectangle-isocèle, et la subdivision est effectuée en le divisant le long du bord formé entre son sommet de sommet et le milieu de son bord de base (voir figure 1.11(e-h)).

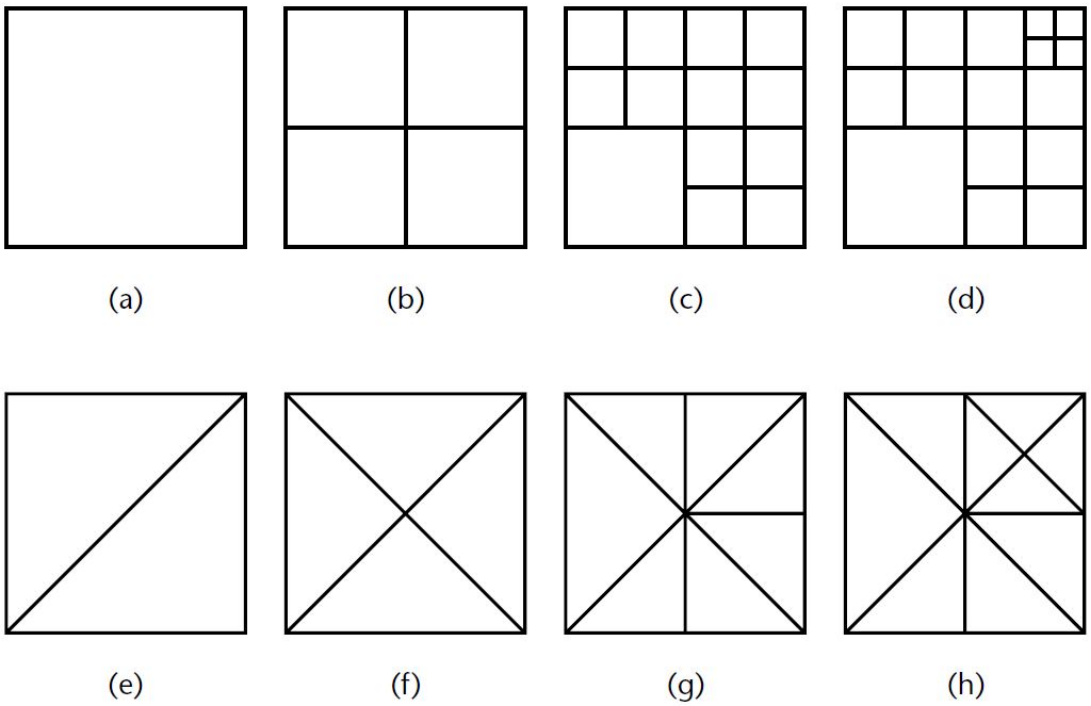


FIGURE 1.11 – Les images (a-d) illustrent le raffinement récursif d'une structure quadratique en forme de carré, et (e-h) illustrent le raffinement récursif d'un arbre binaire triangulaire.

1.7 Bilan

Le tableau comparatif suivant, représente un bilan des algorithmes pour la génération des terrains que nous avons élaborés :

Techniques	Caractéristiques	Avantages	inconvénients
Déplacement médian	-Processus itératif. -La hauteur d'un point dépen des points voisins	- Possibilité de controler le résultat via le coefficient de lisage. -Problème de rainures réglé.	-L'apparence de petites crêtes (sommets) surtout pour les terrain lisses.
Défauts-aléatoire	-Processus itératif. -Une corrution pseudo-aléatoire d'un maillage qui est interprété comme heightmap	-Les plus faciles à mettre en œuvre. -Sa nature itérative permet de l'utiliser dans plusieurs domaines	-Complexité temporelle élevé.
Triangulation de Delaunay	-L'utilisation de la subdivision récursive.	-Possibilité de controler le niveau de détails.	– Des structures adéquates et algorithme simple implique un rendu difficile, – Si on préfère le rendu l'algorithme se complique.
Perlin	-Processus itératif par affinement. -Basé sur les interpolations	-Le résultat est un terrain homogène et réaliste.	-Les interpolations sont très couteuse en temps de calculs.

TABLE 1.1 – Comparaison entre les algorithmes de génération de terrain

1.8 Génération la texture de terrain

Jusqu'à présent, nous nous sommes préoccupés de générer et d'optimiser la géométrie du paysage sans prêter attention aux textures que nous appliquerons à cette géométrie. Ainsi, nous touchons maintenant rapidement l'image satellite, l'génération de texture procédurale, la texturation triplanaire, et enfin,les textures virtuelles

1.8.1 Image satellite

L'une des méthodes les plus simples, et généralement les moins convaincantes, pour appliquer des données de couleur à notre terrain consiste à prendre une seule texture (comme une image satellite ou arienne) et à la recouvrir du réseau de sommets. Le problème avec une telle approche, en supposant d'abord que le rapport d'aspect de l'image correspond à celui du maillage et que les coordonnées de texture appropriées ont été générées, est qu'une image d'un paysage contient des informations que le spectateur s'attend à être en trois dimensions (bâtiments , arbres, etc). Alors qu'un simple terrain cartographié par la texture peut sembler crédible à des milliers de mètres dans l'air, il semble tout à fait peu convaincant au niveau du sol. Les pixels de l'image sont étirés minces à travers les sommets voisins, provoquant un aspect délavé. Les ombres et autres informations d'éclairage sont, bien sûr, statiques dans l'image, ce

qui rend le terrain réaliste uniquement dans les mêmes conditions d'éclairage que celles prises à l'origine, limitant ainsi l'utilité de la visualisation du terrain. Bien que l'imagerie par satellite soit utile pour déterminer la coloration et l'emplacement de la végétation dans le terrain, il est préférable de recréer la surface du terrain avec une texture procédurale en utilisant l'imagerie satellitaire comme fonction de base.

1.8.2 Génération de texture procédurale

La génération de texture procédurale est le processus (procédural) de génération d'une texture. Mis à part les ambiguïtés, c'est un processus dans lequel on peut créer une texture en effectuant des calculs en utilisant des données provenant de diverses sources possibles. Par exemple, les données peuvent être des textures existantes qui seront combinées en une nouvelle texture basée sur un modèle donné, ou des données aléatoires générées par des fonctions de bruit. Dans cette section, nous allons explorer quelques-unes des façons possibles d'utiliser la génération de texture procédurale pour créer des textures personnalisées pour les cartes de hauteur.

1.8.2.1 Combinaison de texture

Un moyen courant de générer une texture personnalisée pour une carte de hauteur consiste à combiner des parties d'autres textures "source" de telle sorte que la texture résultante soit directement liée aux caractéristiques de la carte de hauteur. Par exemple, une texture de neige pourrait être utilisée pour les sommets des montagnes, une texture rocheuse pour les hautes altitudes non recouvertes de neige, une texture d'herbe pour les zones sous les montagnes, une texture de sable pour les zones de plage inférieure et une texture d'eau. points les plus bas sur la carte. il y a deux possibilités pour combiner des textures :

1. Couper des sections de textures et les assembler pour créer une nouvelle texture.
2. Le mélange, une amélioration du découpage, mélange les textures en pondérant les valeurs de couleur des sections en fonction des données de la carte de hauteur.

Exemple : Supposons que notre carte de hauteur a des valeurs de 0 à 255 et que nous combinons 5 textures comme décrit ci-dessus. Nous devons diviser la gamme de la carte de hauteur en 5 régions égales pour représenter chacune des 5 textures.

— Région 1 (eau) : 0-50
— Région 2 (sable) : 51-101
— Région 3 (herbe) : 102-152
— Région 4 (rock) : 153-203
— Région 5 (neige) : 204-255

1.8.2.2 Découpage

Pour générer la texture à l'aide d'un algorithme de découpage, parcourez les valeurs de hauteur (valeurs de pixels) dans la carte de hauteur et sélectionnez la valeur de pixel correspondante dans la texture source dans laquelle se trouve la valeur de hauteur. Par exemple, si la carte de hauteur a une valeur de 115 aux coordonnées (30, 55), alors la valeur de pixel de notre texture générée en (30, 55) sera prise à partir de la valeur de pixel de la texture d'herbe à (30, 55) .

```
1: function GENERER_TEXTURE_AVEC_DECOUPE (heightMap, regions, nouvelleTexture)
2:     Pour chaque (i, j) dans les dimensions de 'heightMap'
3:         h = valeur de pixel de 'heightMap' à (i, j)
4:         t = texture de 'régions' dont la gamme couvre 'h'
5:         v = valeur de pixel de 't' à (i, j)
6:         écrire 'v' à 'nouvelle texture' à la position (i, j)
7: end function
```

1.8.2.3 Mélange

L'algorithme du découpage produit des points de début et de fin notables pour les régions. Une amélioration de cet algorithme consiste à mélanger les valeurs de pixel en fonction de la valeur de hauteur. Toute valeur de hauteur qui correspond à une région $+-$ varie en fonction de cette texture. Plus la valeur de hauteur est proche du sommet (ou de l'extrémité supérieure) d'une région, plus elle sera influencée par cette texture. Pour générer la texture à l'aide d'un algorithme de fusion, parcourez les valeurs de hauteur dans la carte de hauteur et calculez une valeur de pixel pour la texture générée sur la base d'une somme pondérée des valeurs trouvées dans les textures source. Le poids de chaque valeur de pixel dans les textures sources peut être calculé comme suit :

- calculer un poids défini par :

$$1.0 - ((abs(heightValue - max)/range)$$

  Pour une petite optimisation, nous pouvons utiliser un dénominateur commun de 'range' qui échangera une soustraction à virgule flottante avec un entier qui produira :

$$(range - abs(heightValue - max))/range]$$

- return 0 si le poids est inférieur à 0
- Sinon, return la valeur de poids

Par exemple : Utiliser les valeurs d'échantillon ci-dessus avec une valeur de hauteur de 195 aux coordonnées (60, 85) donnerait la somme de :

$$(51 - abs(195 - 50))/51 = -1,843 = 0$$

$$(51 - abs(195 - 101))/51 = -0,843 = 0$$
$$(51 - abs(195 - 152))/51 = 0,157 * Valeurdelatexturedel'herbe(60,85)$$
$$(51 - abs(195 - 203))/51 = 0.843 * Valeurdelatexturederoche(60,85)$$
$$(51 - abs(195 - 255))/51 = -0,176 = 0$$

Notez qu'une texture ne s'appliquera qu'à une hauteur dans la plage upperBound +- exclusivement. Ceci permet une optimisation pour vérifier seulement les deux textures qui affecteront la valeur finale du pixel. Les manières d'appliquer cette optimisation dépendent de l'implémentation. L'algorithme suivant n'inclut aucune optimisation.

```
 1: function GENERER_TEXTURE_AVEC_MELANGE (heightMap, regions, nouvelleTexture)
 2:     Pour chaque (i, j) dans les dimensions de 'heightMap'
 3:         h = valeur de pixel de 'heightMap' à (i, j)
 4:         v = 0
 5:     pour chaque r dans les régions
 6:         v + = poids (h, r) * valeur de pixel de 'r.texture' à (i, j)
 7:
 8:     écrire 'v' à 'nouvelleTexture' à la position (i, j)
 9: end function
10:
 1: function POIDS(hauteur, region)
 2:         w = (region.range - abs (hauteur - region.max)) / region.range
 3:         return (w <0) ? 0 : w
 4: end function
 5:
```

1.8.3 Texturation triplanaire

La texturation triplanaire est une technique basée sur la texturation par projection planaire, qui est la méthode la plus courante de génération de coordonnées de texture pour les terrains basés sur la hauteur. La projection planaire consiste à appliquer une texture sur une surface comme s'il s'agissait d'un quadrilatère. La texturation triplanaire est similaire, sauf que nous pouvons choisir la direction dans laquelle la projection est appliquée en fonction de la surface normale. Essentiellement, nous trouvons quel plan la surface correspond le plus proche du plan x-z, le plan y-z et le plan x-y. Ils appliquent également une texture différente pour chaque projection planaire.

1.8.4 Les textures virtuelles

Les textures virtuelles sont une toute nouvelle forme de texture, qui suit une structure similaire à la mémoire virtuelle. C'est-à-dire que la texture elle-même est stockée dans une mémoire

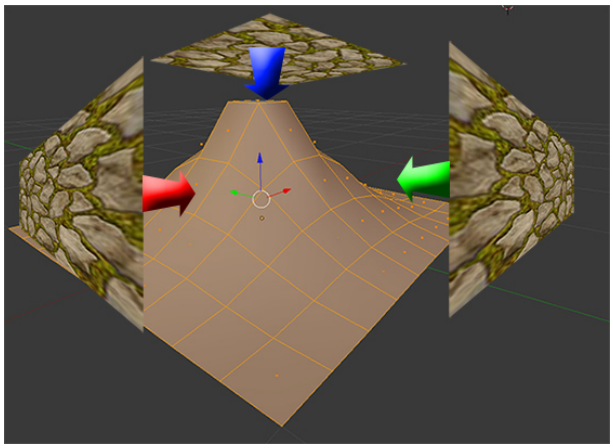


FIGURE 1.12 – Texture projetée à partir de 3 angles.

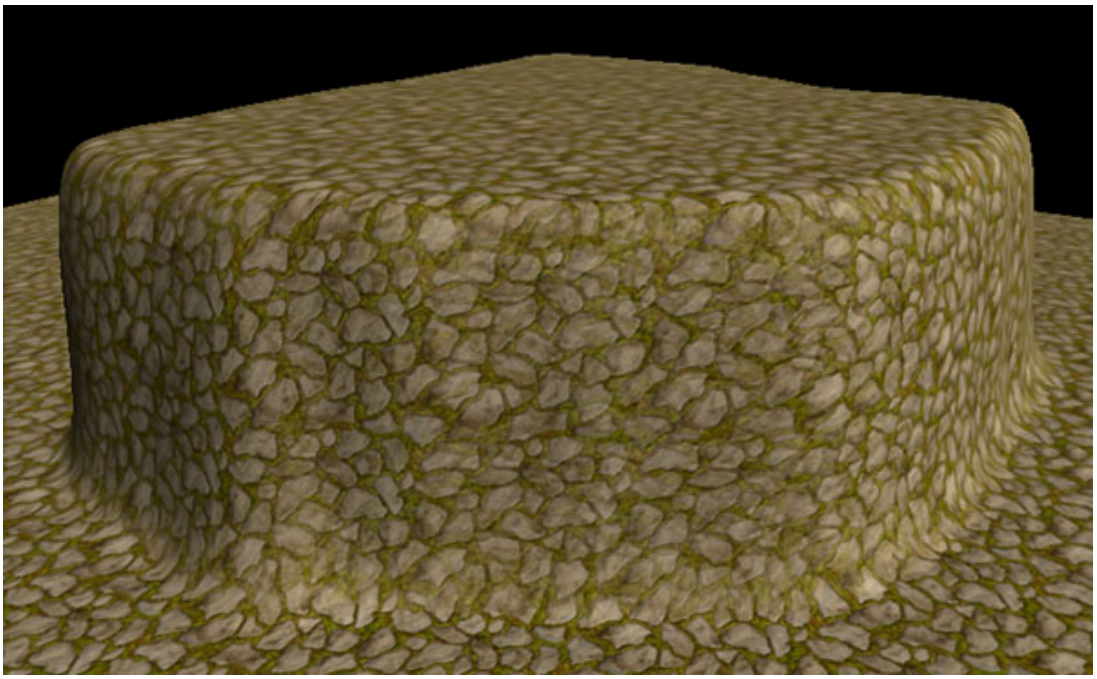


FIGURE 1.13 – Terrain avec texturation triplanaire.

non volatile telle qu'un disque dur et que seules des parties de la texture sont stockées en mémoire à un moment donné. L'avantage de cette méthode est que de très grandes textures peuvent être utilisées, ce qui est idéal pour le rendu du terrain. Une texture virtuelle peut être utilisée pour tout le terrain, ce qui permet aux artistes de prendre le contrôle sans se soucier des limites de taille de la texture.

Bien que cette méthode soit adoptée dans les nouveaux moteurs de jeu, son principal avantage, le contrôle créatif, n'aide pas notre objectif, qui consiste à texturer arbitrairement les heightmaps de façon procédurale. Cela pourrait peut-être améliorer les performances par rapport à d'autres méthodes, mais nous n'avons pas essayé de les mettre en œuvre.

1.9 Conclusion

Ce chapitre est conçu comme une introduction aux concepts de base de la génération de terrain. L'exploration est la clé pour créer des paysages intéressants.

Nous avons examiné quelques algorithmes pour la génération des terrains. Certain de ces algorithmes génèrent des effets indésirables dans le terrain résultat, et certains sont lourd et lent(Algorithme de Perlin).

Nous allons essayer par la suite d'améliorer l'algorithme de Perlin pour la génération de terrain. L'idée de base est de résoudre tous les sous-problèmes dans l'algorithme en utilisant les GPU.

Nous avons ainsi présenté les idées générales sur la génération de textures, et la façon avec la quelle un terrain vas afficher en obtenir un résultat suffisamment réaliste.et la technique la plus utilisée est la technique de mélange de textures de base car elle est plus efficace et plus flexible par sa nature.

Chapitre 2

Aperçu générale de l'infographie

2.1 Introduction

Un pipeline de rendu est la partie la plus importante de nombreux moteurs de jeu, chaque trame, toutes les données de maillage et les shaders doivent être traités pour créer les images pour le jeu lui-même : un pipeline efficace signifie un jeu bien optimisé. Il existe quelques API de pipeline de rendu bien connues : DirectX de Microsoft et Vulkan et OpenGL de l'organisation Khronos.

Cette partie représente, un chapitre introductif à notre travail, où nous présentons certains des concepts de base d'un pipeline de rendu.

Dans ce chapitre aussi nous allons élargir nos connaissances sur la création d'environnements 3D réalistes avec une géométrie simple que possible. Il existe un certain nombre de techniques différentes qui peuvent y parvenir.
Nous allons choisir la meilleure technique à exploiter dans une application en temps réel et l'une de ces techniques (la cartographie de relief).

2.2 OpenGL

OpenGL [16] est une API conçue pour fournir aux développeurs les outils nécessaires à l'écriture d'applications graphiques. Les premières versions utilisaient un pipeline «à fonction fixe» qui restituait les images en utilisant des fonctions par défaut avec des paramètres personnalisables. Bien que le pipeline à fonction fixe soit facile à utiliser, l'approche n'a pas la possibilité de personnaliser le processus de rendu. De plus, l'utilisation de la fonction fixe a tendance à être plus lente en raison de l'utilisation intensive du processeur, ce qui devient un goulot d'étranglement car de grandes quantités de données sont régulièrement transférées vers le GPU.

2.3 Pipeline programmable

Le pipeline programmable est un ajout, introduit dans une version ultérieure d'OpenGL, qui donne aux développeurs un meilleur contrôle du processus de rendu. Les programmes, appelés Shaders, peuvent être écrits en langage GLSL (OpenGL Shading Language), qui sont exécutés à la place du pipeline à fonction fixe. Cela offre aux développeurs une plus grande flexibilité dans le développement d'applications graphiques. Le pipeline programmable utilisé dans les versions précédentes est illustré à la figure 2.1.

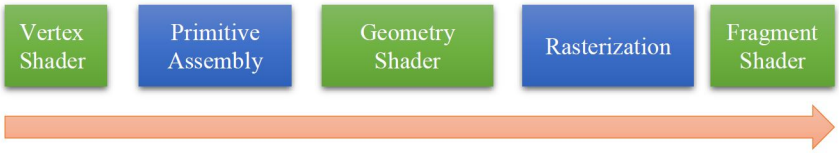


FIGURE 2.1 – Le pipeline graphique avant OpenGL 4. Les étapes programmables sont affichées en vert.

1. **Vertex Shader -**Pour chaque sommet émis par une commande de dessin, un vertex shader sera appelé pour traiter les données associées à ce vertex. Selon que d'autres shaders pré-rasterisation sont actifs, les vertex shaders peuvent être très simples, peut-être simplement copier des données pour passer à travers cette étape shaders — ce que nous appellerons un shader pass-through à un très complexe shader qui effectue de nombreux calculs pour potentiellement calculer la position de l'écran du vertex (généralement en utilisant des matrices de transformation), déterminer la couleur du vertex en utilisant les calculs d'éclairage décrits ou toute autre technique. En règle générale, une application de toute complexité aura plusieurs vertex shaders, mais un seul peut être actif à la fois.
2. **Primitive Assembly -** Les étapes des shaders précédentes fonctionnent toutes sur des sommets, avec les informations sur la façon dont ces sommets sont organisés en primitives géométriques transportées de manière interne vers OpenGL. L'étape d'assemblage primitif organise les sommets en leurs primitives géométriques associées en vue de l'écrêtage et de la pixellisation.
3. **Geometry Shading -** L'étape de shader suivante — geometry shading — permet un traitement supplémentaire des primitives géométriques individuelles, y compris la création de nouvelles, avant la pixellisation. Cette étape d'ombrage est également facultative, mais très puissante.
4. **Rasterization -** les primitives mises à jour sont envoyées au rasterizer pour la génération de fragments. Considérons un fragment comme un «pixel candidat», en ce sens que les pixels ont une origine dans le framebuffer, tandis qu'un fragment peut toujours être rejeté et ne met jamais à jour son emplacement de pixel associé. Le traitement des fragments

se produit dans les deux étapes suivantes, fragment shading et les opérations par per-fragment.

5. **Fragment Shading -** L'étape finale où vous avez un contrôle programmable sur la couleur d'un emplacement d'écran est pendant *fragment shader*. Dans cette étape de shader, vous utilisez un shader pour déterminer la couleur finale du fragment (bien que la prochaine étape, les opérations par fragment puissent modifier la couleur une dernière fois), et potentiellement sa *valeur de profondeur*. Les fragments Shaders sont très puissants car ils utilisent souvent un mappage de texture pour augmenter les couleurs fournies par les étapes de traitement de vertex. Un fragment shader peut également terminer le traitement d'un fragment s'il détermine que le fragment ne doit pas être dessiné ; ce processus est appelé *fragment discard*.

2.4 OpenGL 4 pipeline

OpenGL 4 est une API récente [**?**] qui permet aux développeurs de cibler le matériel Direct3D 11 [1]. Les unités de traitement graphique modernes (GPU) offrent des fonctionnalités qui n'étaient auparavant pas disponibles pour les développeurs. L'ajout le plus significatif est le pipeline OpenGL modifié, qui introduit un étage de tessellation, visible sur la figure 2.2. L'étage de tessellation est utilisé pour subdiviser un polygone donné et appliquer certaines transformations. Cette fonctionnalité peut être appliquée efficacement à un modèle 3D pour obtenir un LOD dynamique très facilement. L'étape de tessellation est utilisée par le développeur en écrivant deux programmes de shaders : un shader de contrôle de tessellation et un shader d'évaluation de tessellation. Le shader de contrôle de tessellation est utilisé pour sélectionner par programme une tessellation appropriée.

FIGURE 2.2 – Le pipeline récemment introduit dans OpenGL 4. Les étapes programmables sont indiquées en vert.

2.5 Tessellation Shader

Une fois que le vertex shader a traité les données associées à chaque sommet, l'étape de shader de tessellation continuera à traiter ces données, si elle a été activée.

1. Direct3D est une API alternative à OpenGL, qui est couramment utilisée comme standard par les fournisseurs de matériel.

La tessellation utilise des patchs pour décrire la forme d'un objet, et permet des collections relativement simples de géométrie de patch pour augmenter le nombre de primitives géométriques fournissant des modèles plus beaux. L'étape de tessellation shader peut potentiellement utiliser deux shaders pour manipuler les données de patch et générer la forme finale.

2.5.1 Tessellation Control Shaders

Une fois que votre application émet un correctif, le shader de contrôle de tessellation sera appelé (s'il y a un lien) et est responsable de l'exécution des actions suivantes :

- Générer les sommets de patch de sortie de tessellation qui sont passés au shader d'évaluation de tessellation, ainsi que mettre à jour tout par vertex ou des valeurs d'attribut par patch si nécessaire.
- Spécifiez les facteurs de niveau de tessellation qui contrôlent le fonctionnement du générateur de primitives. Ce sont des variables de shader de contrôle de tessellation spéciales appelées *gl_TessLevelInner* et *gl_TessLevelOuter*, et sont implicitement déclarées dans votre shader de contrôle de tessellation.

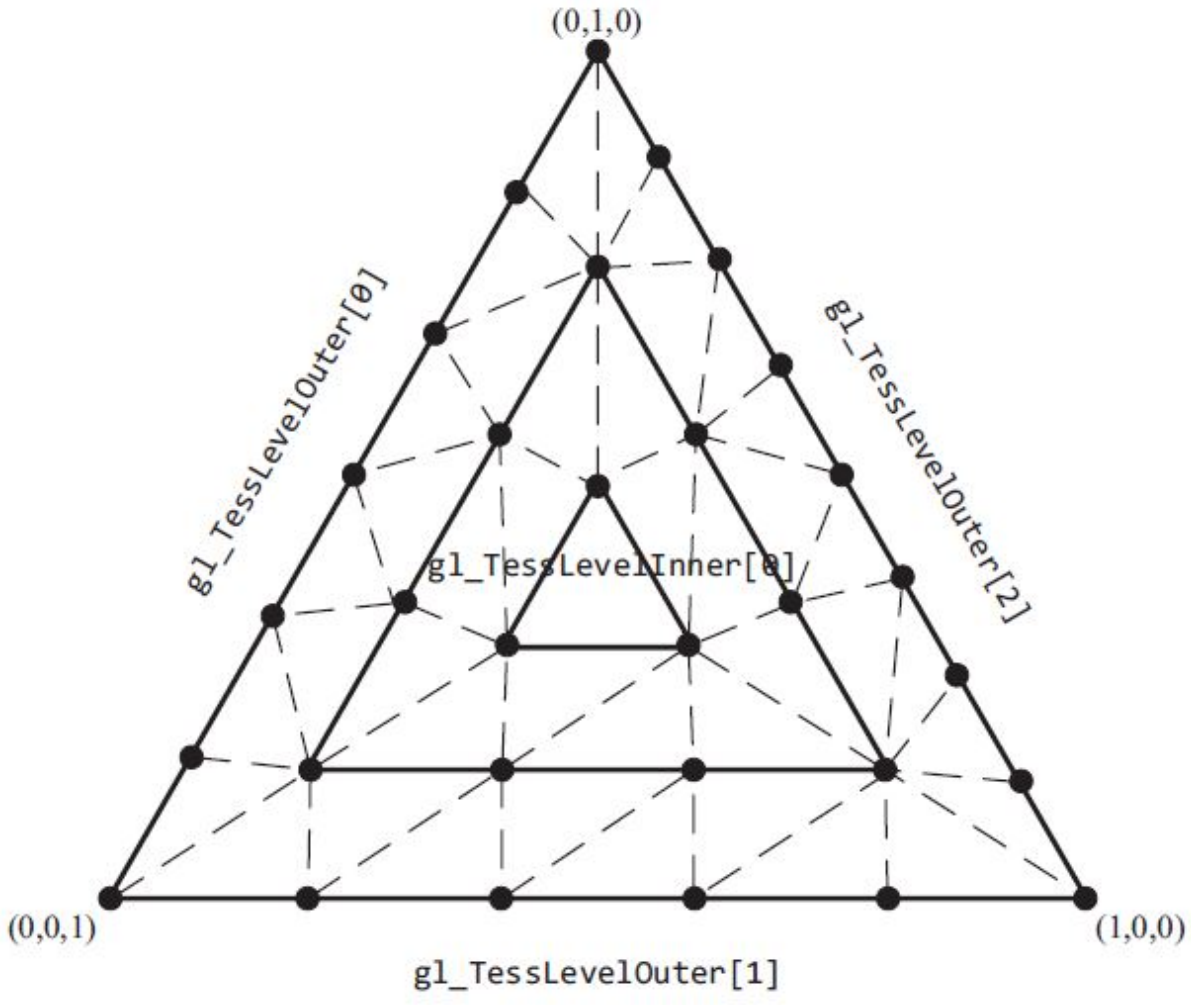


FIGURE 2.3 – Tessellation de triangle.

2.5.2 Tessellation Evaluation Shaders

La phase finale dans le pipeline de tessellation d'OpenGL est l'exécution de shader d'évaluation de tessellation. Le shader d'évaluation de la tessellation liée est exécuté pour chaque coordonnée de tessellation que le générateur de primitives émet et est responsable de la détermination de la position du sommet dérivé de la coordonnée de tessellation. Comme nous le verrons, les shaders d'évaluation de tessellation ressemblent à des vertex shaders pour transformer des sommets en positions d'écran (à moins que les données du shader d'évaluation de tessellation ne soient traitées par un shader de géométrie).

La première étape de la configuration d'un shader d'évaluation de tessellation est de configurer le générateur de primitives, ce qui est fait en utilisant une directive **layout**, similaire à ce que nous avons fait dans le shader de contrôle de tessellation. Ses paramètres spécifient le domaine de tessellation et par la suite, le type de primitives générées ; l'orientation du visage pour les primitives solides (utilisées pour l'abattage du visage) ; et comment les niveaux de tessellation devraient être appliqués pendant la génération primitive.

2.6 Techniques de relief

Toutes les techniques de relief ont le même objectif qui est la simulation des délais dans un surface (relief) mais chaque technique possède sa propre façon d'implémentation nous pouvons voir la différence dans les résultats et chaque technique elle a des limites.

2.6.1 Normal Mapping

FIGURE 2.4 – Démonstration de la technique de Normal Mapping. Sur la gauche, la cartographie normale devant la caméra, la figure sur la droite montre l'inconvénient de Normal Mapping.

Normal Mapping [17] est peu coûteuse et facile à mettre en œuvre. Pour réaliser des bosses avec cette technique, une texture avec des décalages de normales est requise. La texture est illustrée à la figure 2.5.

L'objet en cours de dessin a ses normales face à une direction de polygone, ce qui rend l'objet plat. Pour créer des bosses, vous changez les normales de pixel en échantillonnant la texture

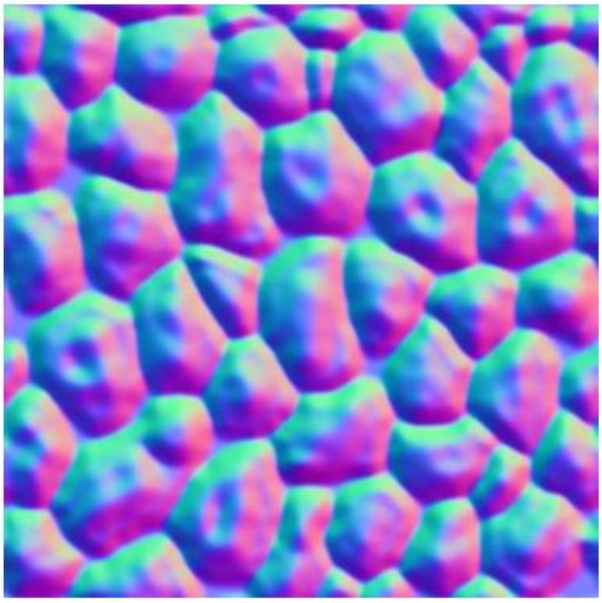


FIGURE 2.5 – Normales décalées stockées dans une texture et mappées sur la plage [0, 1].

de normale et en transformant la normale actuelle avec la texture échantillonnée.
Le normal mapping n'affecte que les normales, donc exposé aux lumières, l'objet semble être plus détaillé qu'il ne l'est en réalité. Cette technique présente un inconvénient, lorsque l'on regarde des objets à partir d'un angle oblique, l'illusion disparaît car le normal mappingl ne déplace pas les coordonnées de texture. La technique de cartographie normale et son inconvénient sont illustrés à la figure 2.4.

2.6.2 Relief Mapping

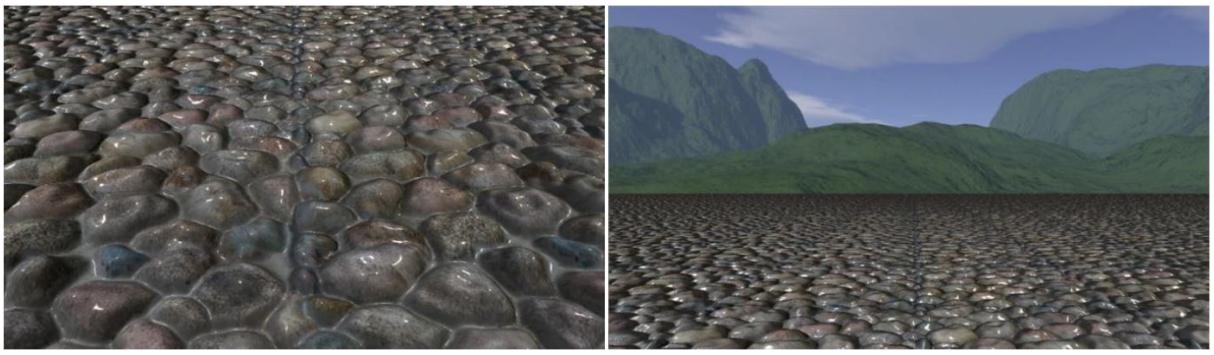


FIGURE 2.6 – Démonstration de relief mapping.

La plus grande différence entre normal mapping et la relief mapping [18] est que relief mapping déplace les coordonnées de la texture pour créer de véritables bosses, comme le montre la figure 2.6. Pour réaliser ce déplacement, une nouvelle texture est nécessaire. Alors que relief mapping a une texture normal map, elle nécessite également une texture de carte de hauteur, illustrée à la figure 2.7.

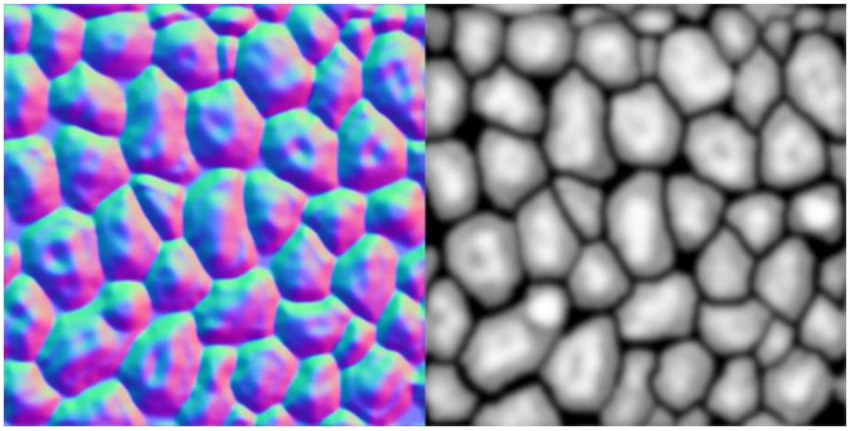


FIGURE 2.7 – A gauche est la texture normal mapping et à droite la texture de height map, les deux sont stockés dans la plage [0, 1].

Height map montre la profondeur de la texture et il est utilisé pour déplacer correctement les coordonnées de la texture.
Une fois la coordonnée de texture calculée, nous calculons l'éclair de la même manière que le normal mapping.

Relief mapping utilise un biais de profondeur pour retirer les artefacts. La polarisation de profondeur bloque le vecteur de décalage de sorte que la taille du pas pour le calcul du relief ne soit pas trop grande. Si la taille du pas est trop grande, la recherche linéaire manquera son point d'intersection et en obtiendra une incorrecte. Cela crée des artefacts, comme illustré dans la figure 2.8. Ce biais de profondeur fournit un déplacement sans artefact, mais il aplatit également la profondeur à distance. Avec ce biais de profondeur, la cartographie de relief n'extrude pas autant de profondeur de loin que Parallax Occlusion Mapping. Cela donne au relief une cartographie presque égale des performances sur tous les angles.

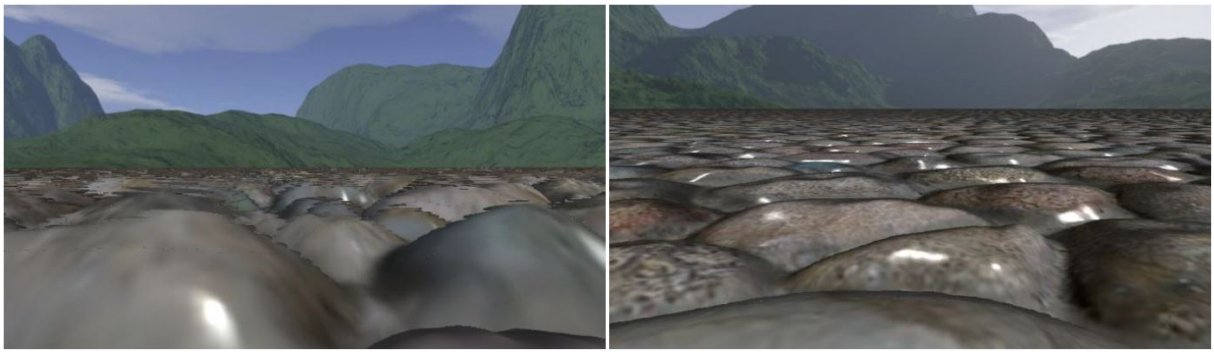


FIGURE 2.8 – A gauche est la cartographie en relief sans biais de profondeur et à droite est en biais de profondeur.

2.6.3 Parallax Occlusion Mapping

Parallax Occlusion Mapping [19] utilise une approche légèrement différente lorsqu'il s'agit de déplacer les coordonnées de texture. Au lieu d'utiliser une combinaison d'une recherche

linéaire et d'une recherche binaire, l'occlusion de parallaxe n'utilise qu'une recherche linéaire et une approximation du profil de hauteur comme une courbe linéaire pas à pas. Comme illustré à la figure 2.10.

FIGURE 2.9 – Parallax occlusion mapping.

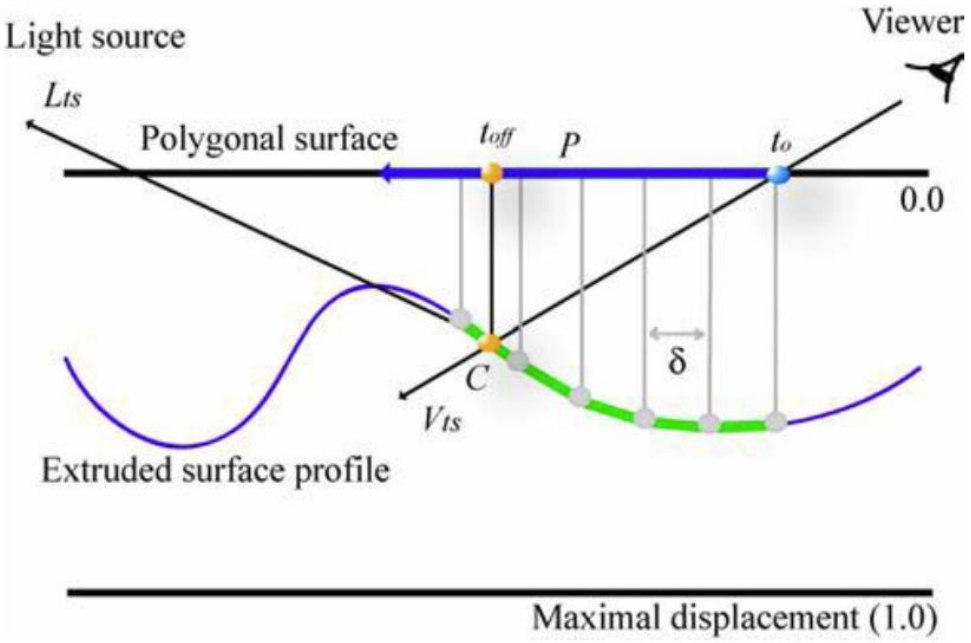


FIGURE 2.10 – Déplacement basé sur le champ de hauteur échantillonné et la direction de la vue actuelle .

2.6.4 Displacement mapping

Les Displacement maps [20] sont parfois utilisées pour changer l'emplacement des sommets réels dans un maillage. Ce type de déplacement n'ajoute aucun détail supplémentaire. Au lieu de cela, il est utilisé pour générer des objets autrement complexes. Un exemple de ce type de déplacement est la manière dont le terrain est souvent généré à partir d'une texture.

Une autre utilisation pour displacement map est le parallax mapping (aussi appelée displacement mapping virtuelle), qui est une technique plus avancée dans laquelle un moteur de

jeu tente de décaler les coordonnées de texture à l'écart de la caméra. C'est assez coûteux en termes de calcul, mais cela peut donner de bons résultats.

FIGURE 2.11 – Démonstration de displacement mapping.

2.7 Bilan

Le tableau comparatif suivant, représente un bilan des techniques pour simuler les détails dans un surface (relief) :

Techniques	Caractéristiques	Avantages	inconvénients
Normal Mapping	Changer les normales de pixel par les normales du Normal map	Peu coûteuse et facile à mettre en	Manque de réalisme (pas de modification sur la géométrique du l'objet).
Relief Mapping	Déplacée les coordonnées de la texture (par normal map et height map) pour créer de véritables bosses	Le résultat est plus réaliste	-Temps de calculer est élevé. -Manque biais de profondeur.
Parallax Occlusion Mapping	Déplacée les coordonnées de la texture et elle utilise une recherche linéaire et approximation du profil de hauteur	-Le résultat est réel. - Elle ajoute biais de profondeur.	-Temps de calculer est élevé. -Offre une approche légèrement plus complexe. -Une forme plus complexe de Relief Mapping.
Displacement mapping	Perturbation du maillage a partir une heightmap	-Augmente le niveau de détails. -Le résultat est plus réaliste	-Temps de calculer est élevé par rapport de normal map.

TABLE 2.1 – Comparaison entre les techniques de relief

2.8 Conclusion

Dans ce chapitre nous avons essayer de donner un aperçu générale de l'infographie, qui concerne certains concepts de base d'un pipeline de rendu et la création creux et de bosses dans les objets 3D,en mettant l'accent sur les idées et les concepts que nous avons utilisés.
Et nous avons découvert qu'il était possible d'optimiser ces techniques. Les différentes techniques de cartographie de bosses que nous avons étudiées ont montré qu'elles sont différentes en termes de performances et de résultats visuels alors qu'elles sont basées sur la même idée, pour créer une véritable profondeur sous tous les angles.

Chapitre 3

Implémentation et Résultats

3.1 Conception de l'application

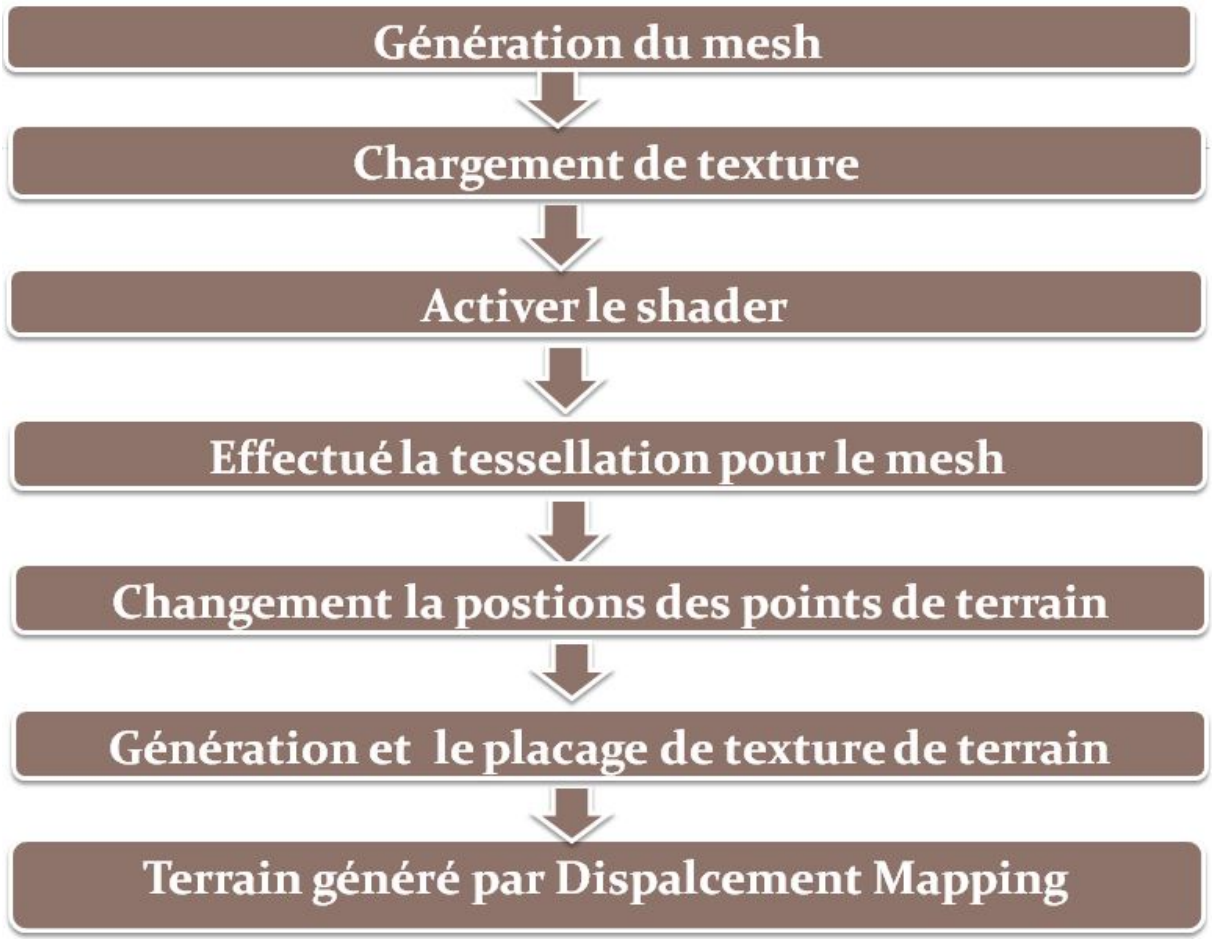


FIGURE 3.1 – Schéma général de l'application .

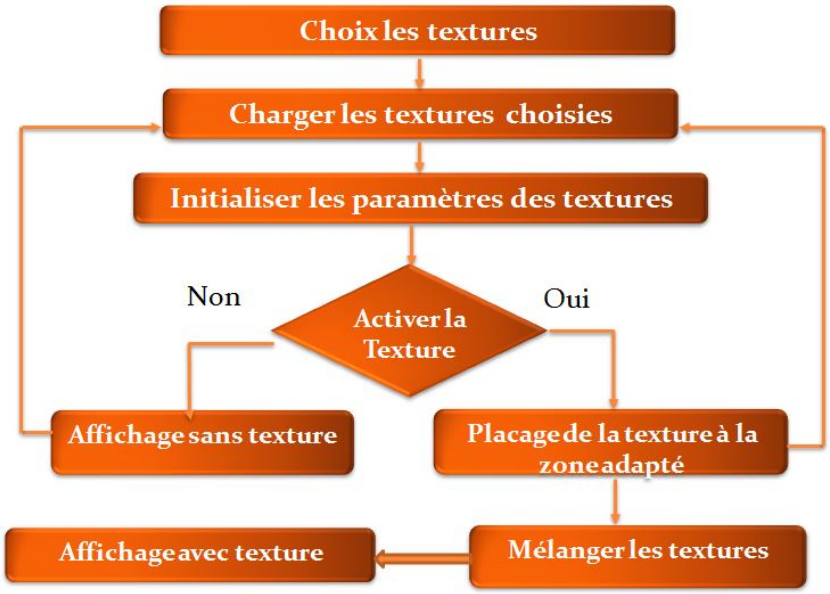


FIGURE 3.2 – Schéma de placage de texture.

3.2 Implémentation

3.2.1 Environnement de développement

L'algorithme a été implémenté sur un PC fonctionnant sous Windows 10 64 bits, avec un processeur Intel (R) Core (TM) i5-6300HQ à 2,30 GHz, 6 Go de RAM et une carte graphique NVIDIA GeForce GTX 950M, en utilisant Microsoft Visual Studio 2010. Nous avons utilisé les bibliothèques suivantes : l'API OpenGL, la boîte à outils FreeGlut, les mathématiques OpenGL (GLM) et l'extension d'extension OpenGL (GLEW). L'implémentation de l'algorithme consiste en une classe C ++ et cinq programmes shader utilisant toutes les étapes du pipeline OpenGL 4.

3.2.2 Génération de terrain

Displacement map est une texture qui contient le déplacement d'une surface à chaque emplacement. Chaque patch représente une petite région d'un paysage qui est subdivisée en fonction de l'espace d'écran. Chaque sommet tessellé est déplacé le long de la tangente à la surface par la valeur stockée dans la carte de déplacement. Cela ajoute des détails géométriques à la surface sans avoir besoin de stocker explicitement les positions de chaque sommet tessellé. Au contraire, seuls les déplacements d'un paysage par ailleurs plat sont stockés dans la carte de déplacement et sont appliqués à l'exécution dans le shader d'évaluation de pavage. Displacement map (également appelée carte de hauteur) utilisée dans le projet est illustrée à la figure 3.13.

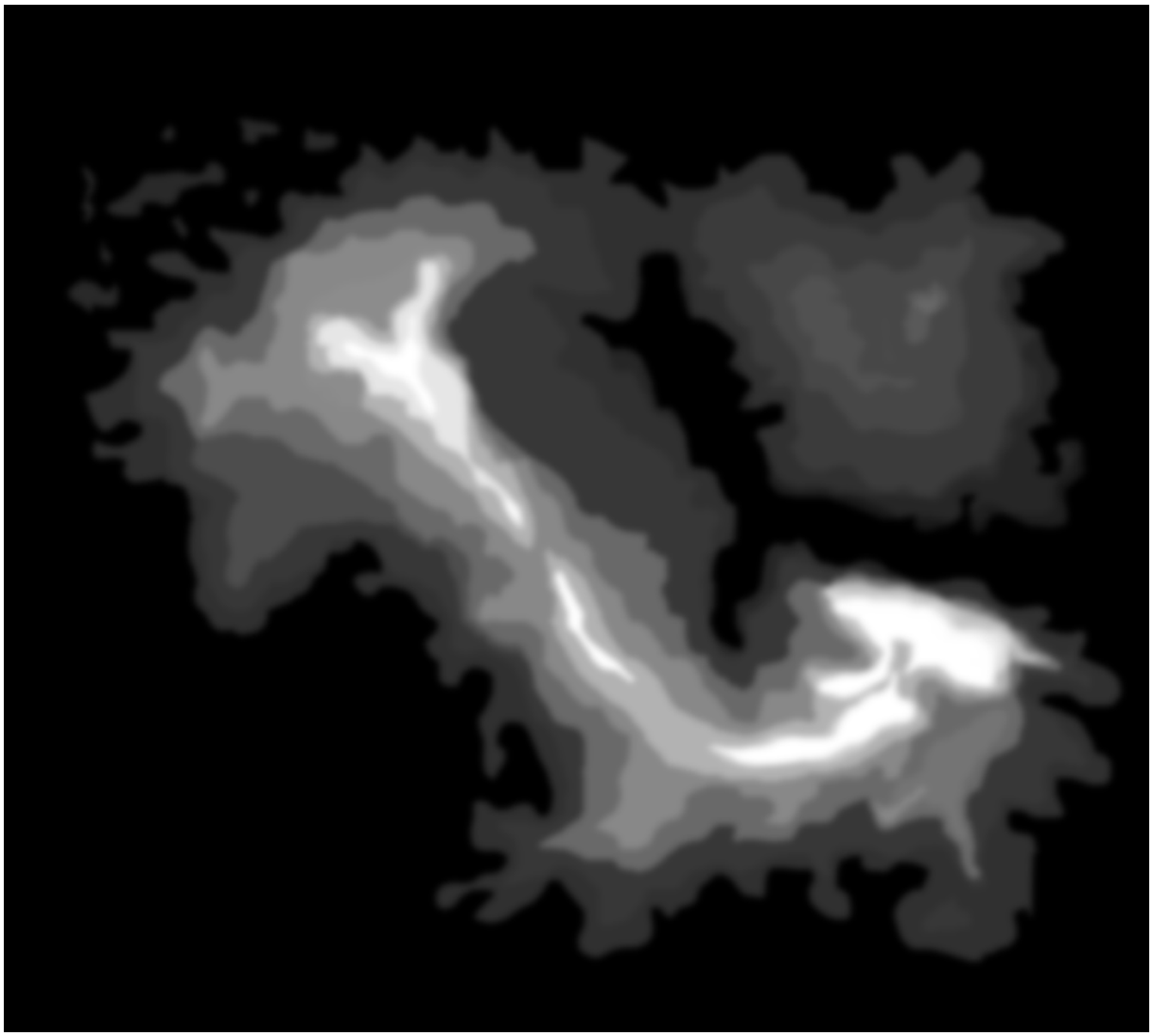


FIGURE 3.3 – Displacement map utilisée dans le terrain.

Notre première étape consiste à configurer un vertex shader. Comme chaque patch est effectivement un quad simple. Le shader complet est montré dans Listing 3.1. Le shader utilise le numéro d'instance (stocké dans ***gl_InstanceID***) pour calculer un décalage pour le patch, qui est un carré d'une unité dans le plan xz, centré sur l'origine. Dans ce projet, nous allons calculer une grille de 64 * 64 patchs, et les décalages x et y pour le patch sont calculés en prenant *gl_InstanceID modulo 64* et *gl_InstanceID divisés par 64*. Le vertex shader calcule également les coordonnées de texture du patch, qui sont passées au shader de contrôle de tessellation dans *vs_out.tc*.

```
#version 430 core
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;
out VS_OUT
{
vec2 tc;
vec3 normal;
} vs_out;

void main(void)
{
  int x = gl_InstanceID & 63;
  int y = gl_InstanceID >> 6;
  vec2 offs = vec2(x, y);

  vs_out.normal=normal+ vec3(float(x - 32), 0.0,float(y - 32));
  vs_out.normal.xz=(normal.xz+ offs + vec2(1.0))/64;

  vs_out.tc=(position.xz+ offs + vec2(1.0))/64;
  gl_Position =  vec4(position,1)+ vec4(float(x - 32),
  0.0,float(y - 32), 0.0);

}
```

Listing 3.1 – Vertex shader pour le rendu du terrain.

Ensuite, nous arrivons au shader de contrôle de la tessellation. Encore une fois, le shader complet est montré dans le Listing 3.2. La majeure partie de l'algorithme de rendu est implémentée dans le shader de contrôle de tessellation, et la majorité du code n'est exécutée que par le premier invocation. Une fois que nous avons déterminé que nous sommes la première invocation en vérifiant que gl_InvocationID est nul, nous calculons les niveaux de tessellation pour le patch entier.

Lorsque le mode tessellation est défini sur des triangles (à nouveau, en utilisant un qualificateur de disposition d'entrée dans le shader de commande de tessellation), le moteur de pavage produit un triangle qui est ensuite divisé en plusieurs triangles plus petits. Seul le premier élément du tableau *gl_TessLevelInner []* est utilisé, et ce niveau est appliqué à l'intégralité de la zone interne du triangle tessellé. Les trois premiers éléments du tableau *gl_TessLevelOuter []* sont utilisés pour définir les facteurs de tessellation pour les trois arêtes du triangle.
Le shader de contrôle de tessellation utilisé pour générer la figure 3.6 est montré dans Listing 3.2.

```
#version 430 core
layout(vertices = 3) out;

in VS_OUT
{
vec2 tc;
vec3 normal;
} tcs_in[];

out TCS_OUT
{
vec2 tc;
vec3 normal;
} tcs_out[];

void main(void)
{
 if (gl_InvocationID == 0)
 {
        gl_TessLevelInner[0] =16.0;
        gl_TessLevelOuter[0] =16.0;
        gl_TessLevelOuter[1] =16.0;
        gl_TessLevelOuter[2] =16.0;
 }
 gl_out[gl_InvocationID].gl_Position=gl_in[gl_InvocationID].gl_Position;
 tcs_out[gl_InvocationID].tc = tcs_in[gl_InvocationID].tc;
 tcs_out[gl_InvocationID].normal = tcs_in[gl_InvocationID].normal;
}
```

Listing 3.2 – Tessellation control shader pour le rendu du terrain.

Une fois que le shader de contrôle de tessellation a calculé les niveaux de tessellation pour le patch, il copie simplement son entrée à sa sortie. Il le fait par instance et transmet les données résultantes au shader d'évaluation de la tessellation, qui est montré dans le Listing 3.3.

```
#version 430 core

uniform sampler2D heightmap;
uniform sampler2D dirt_depth;
uniform sampler2D rock_depth;
uniform mat4 MVP;
layout (triangles, equal_spacing, ccw) in;

in TCS_OUT
{
vec2 tc;
vec3 normal;
} tes_in[];
out TES_OUT
{
vec2 tc;
vec3 normal;

} tes_out;
out vec3 WorldPos_FS_in;

vec2 interpolate2D(vec2 v0, vec2 v1, vec2 v2)
{
return vec2(gl_TessCoord.x) * v0 + vec2(gl_TessCoord.y) * v1
 + vec2(gl_TessCoord.z) * v2;
}
vec3 interpolate3D(vec3 v0, vec3 v1, vec3 v2)
{
return vec3(gl_TessCoord.x) * v0 + vec3(gl_TessCoord.y) * v1
+ vec3(gl_TessCoord.z) * v2;
}

void main(void)
{
float a=0.5,b=1.5,c=4.5,d=5.5,e=7,f=9;

tes_out.tc = interpolate2D(tes_in[0].tc,tes_in[1].tc,tes_in[2].tc);
tes_out.normal=interpolate3D(tes_in[0].normal,tes_in[1].normal
                ,tes_in[2].normal);
WorldPos_FS_in =interpolate3D(gl_in[0].gl_Position.xyz,
                 gl_in[1].gl_Position.xyz, gl_in[2].gl_Position.xyz);

WorldPos_FS_in.y += texture(heightmap, tes_out.tc).x * 11;

gl_Position = MVP * vec4(WorldPos_FS_in,1);
}
```

Listing 3.3 – Tessellation evaluation shader pour le rendu du terrain.

Le shader d'évaluation de la tessellation calcule d'abord la coordonnée de texture du vertex généré en interpolant linéairement les coordonnées de texture transmises par le shader de contrôle de tessellation du Listing 3.2 (généré à son tour par le vertex shader du Listing 3.1). Il applique ensuite une interpolation similaire aux positions du point de contrôle entrant pour produire la position du sommet sortant. Cependant, une fois cela fait, il utilise la coordonnée de texture calculée pour décaler le sommet dans la direction y avant de multiplier ce résultat par la matrice model-view-projection (la même que celle utilisée dans le shader de contrôle de tessellation). Il transmet également la coordonnée de texture calculée au fragment shader dans ***tes_out.tc***. Ce fragment de shader est montré dans le Listing 3.4.

```
#version 430 core
out vec4 FragColor;

in TES_OUT
{
vec2 tc;
} fs_in;
void main(void)
{
FragColor =vec4(0.0,1.0,0.0,1.0);
}
```

Listing 3.4 – Fragment shader pour le rendu du terrain.

Le résultat du rendu avec cet ensemble de shaders est montré dans la figure 3.6 ,si vous regardez la version filaire de l'image montrée à la figure 3.6 , vous pouvez voir clairement le maillage triangulaire sous-jacent du paysage. Les buts du programme sont que tous les triangles restitués à l'écran ont une surface d'écran similaire et que les transitions pointues au niveau de la tessellation ne sont pas visibles dans l'image rendue.

3.2.3 Génération de texture de terrain

Dans cette partie en va parler de colorisation les différentes zones du terrain, l'idée est de générer une texture qui sera faite parfaitement par rapport au terrain qui sera affiché. Ce que nous voulons faire est d'utiliser plusieurs textures différentes pour afficher correctement le terrain, la solution est appelée "génération de texture procédurale".

Les texture sont ce que vous voulez qu'ils soient, mais dans notre cas ils représentent différents attributs de notre paysage (Eau, sol, roches, roches avec de la glace, neige, etc.)

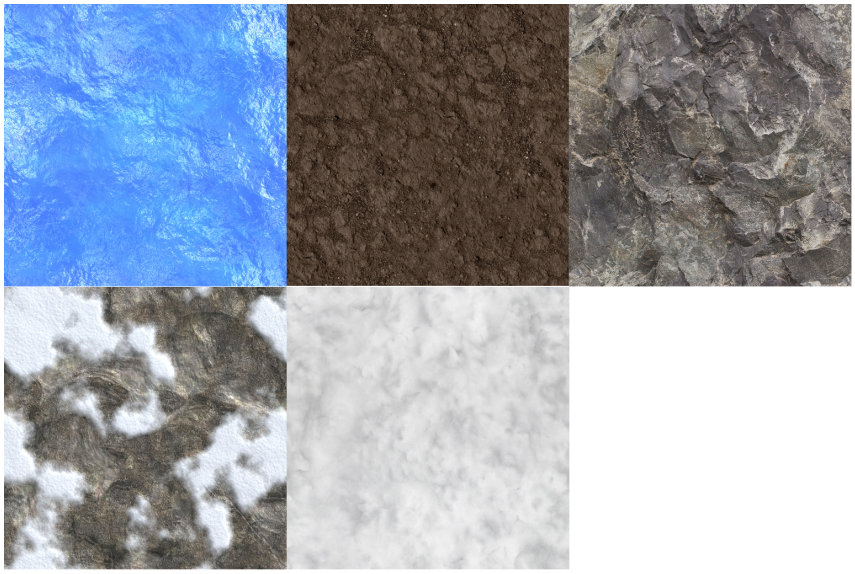


FIGURE 3.4 – Textures de base.

– Nous faisons maintenant ce qui suit :

1. Lire le pixel à l'emplacement (x / y) dans heightmap et évaluez la valeur de la hauteur.
2. Effectuez l'étape 1 pour tous les autres textures que nous avons.

3.2.3.1 Calculer la hauteur h

Pour calculer la hauteur du pixel associé, on utilise une heightmap. Si nous sommes dans le meilleur des mondes, la taille de la texture du terrain est la même que heightmap. C'est donc facile d'avoir la hauteur du pixel associé.

3.2.3.2 Utiliser la valeur h pour calculer la couleur du pixel

Pour avoir la couleur du pixel, nous utilisons directement la hauteur du pixel. Dépendant de la hauteur, nous allons utiliser un mélange entre les différentes images de niveaux. Si la hauteur est assez basse, on utilisera uniquement la couleur de l'image qui représente du sol. Ensuite, si la hauteur est très haute alors on utilise la couleur de l'image de neige. Bien sûr, cela veut dire

que nous pouvons faire des mélanges entre les cinq textures.Ce code est montré dans le Listing 3.5.

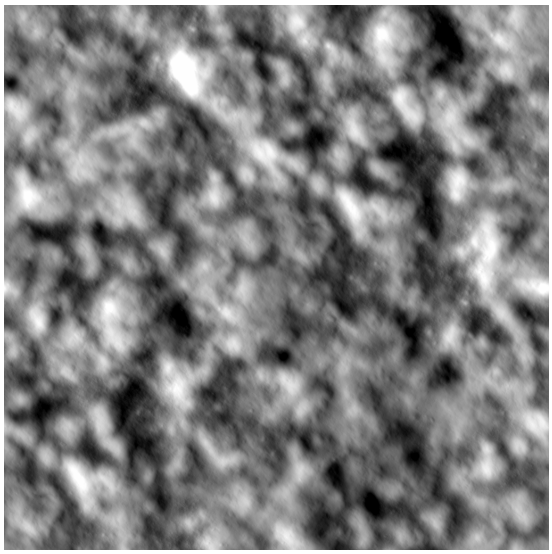
```
#version 430 core
out vec4 FragColor;

in TES_OUT
{
 vec2 tc;
} fs_in;

void main(void)
{
 float a=0.5,b=1.5,c=4.5,d=5.5,e=7,f=10;
   if(WorldPos_FS_in.y<=a)
        {
        FragColor = mix(texture(dirt, fs_in.tc*8),
                    texture(water, fs_in.tc*8),0.5);
        }else{

          if(WorldPos_FS_in.y<b)
           {
            float tLocal1 = 1-( WorldPos_FS_in.y - a ) / ( b - a );
            float tLocal2 = ( WorldPos_FS_in.y - a ) / ( b - a );

            if(WorldPos_FS_in.y<(b-0.8)){

             FragColor  += mix((texture(dirt, fs_in.tc*8)*tLocal1)
             +(texture(rock, fs_in.tc*8)*tLocal2),
             texture(water, fs_in.tc*8),0.5);

             }else{

             FragColor  += (texture(dirt, fs_in.tc*8)*tLocal1)
             +(texture(rock, fs_in.tc*8)*tLocal2);

                 }

             }else{
```

```
 if(WorldPos_FS_in.y<c)
{

FragColor += texture(rock, fs_in.tc*8);

}else{

if(WorldPos_FS_in.y<d){

  float tLocal1 =1-( WorldPos_FS_in.y - c) / ( d - c);
  float tLocal2 = ( WorldPos_FS_in.y - c) / ( d - c);
  FragColor+=(texture(rock,fs_in.tc*8)*tLocal1)+
  (texture(rock_snow, fs_in.tc*8)*tLocal2);

} else{

if(WorldPos_FS_in.y<e)
{
 FragColor += texture(rock_snow, fs_in.tc*8);
}else{

          if(WorldPos_FS_in.y<f)
          {
           float tLocal1 =1-( WorldPos_FS_in.y - e) / ( f - e);
           float tLocal2 = ( WorldPos_FS_in.y - e) / ( f - e);
           FragColor+=(texture(rock_snow,fs_in.tc*8)*tLocal1)
           +(texture(snow,fs_in.tc*8)*tLocal2);

          }else{
                  FragColor += texture(snow, fs_in.tc*8);
              }
}
}
}

}
}
```

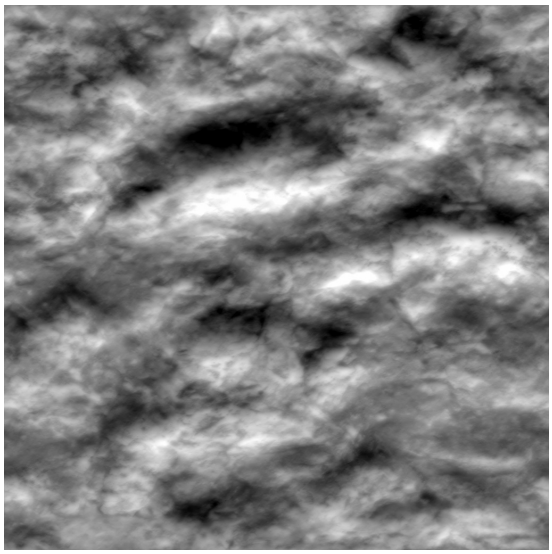
Listing 3.5 – Fragment shader pour le rendu du terrain avec la texture.

3.2.3.3 Textures détaillées : ajout de détails

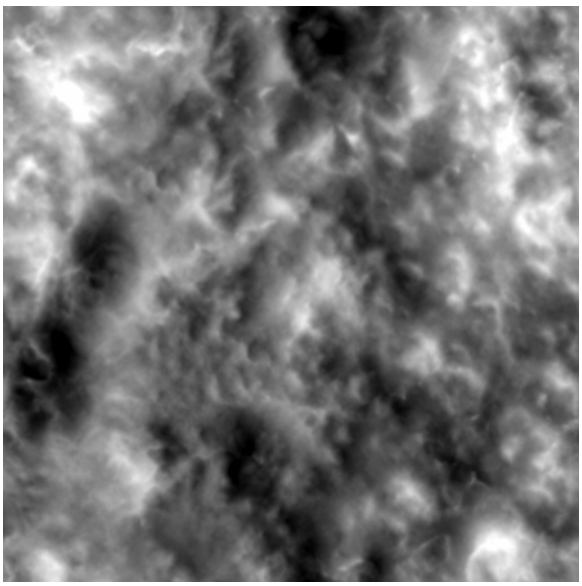
Les cartes de détail sont généralement fabriqués à partir de textures avec beaucoup de détails et sont mis en correspondance avec les polygones de terrain un à un la plupart du temps. La texture utilisée pour la carte détaillée doit être supprimée en effectuant une fonction de désaturation dans Photoshop ou tout autre outil d'édition de texture afin d'obtenir une image en échelle de gris, comme ceux-ci :



(a) Sol



(b) Roches



(c) Neige

FIGURE 3.5 – Textures détaillées.

En vas utilise ces textures pour ajouter beaucoup plus de détails sur le terrain. Elles seront répétées un certain nombre de fois sur tout le paysage.

3.3 Resultats

Dans cette partie nous allons présenter un ensemble de résultats permettant de valider les différentes méthodes proposées dans ce travail, ces résultats sont obtenus en utilisant un ordinateur qui possè de les caractéristiques suivantes :

— Windows 10 64 bits
— un processeur Intel (R) Core (TM) i5-6300HQ à 2,30 GHz
— 6 Go de RAM
— une carte graphique NVIDIA GeForce GTX 950M
— version OpenGL 4.4

3.3.1 Génération de terrain

Dans cette application, nous allons calculer une grille de 64 * 64 patchs et nous calculons les niveaux de tessellation pour l'ensemble du patch.
Et voici une image qui montre un terrain affiché en mode filaire :

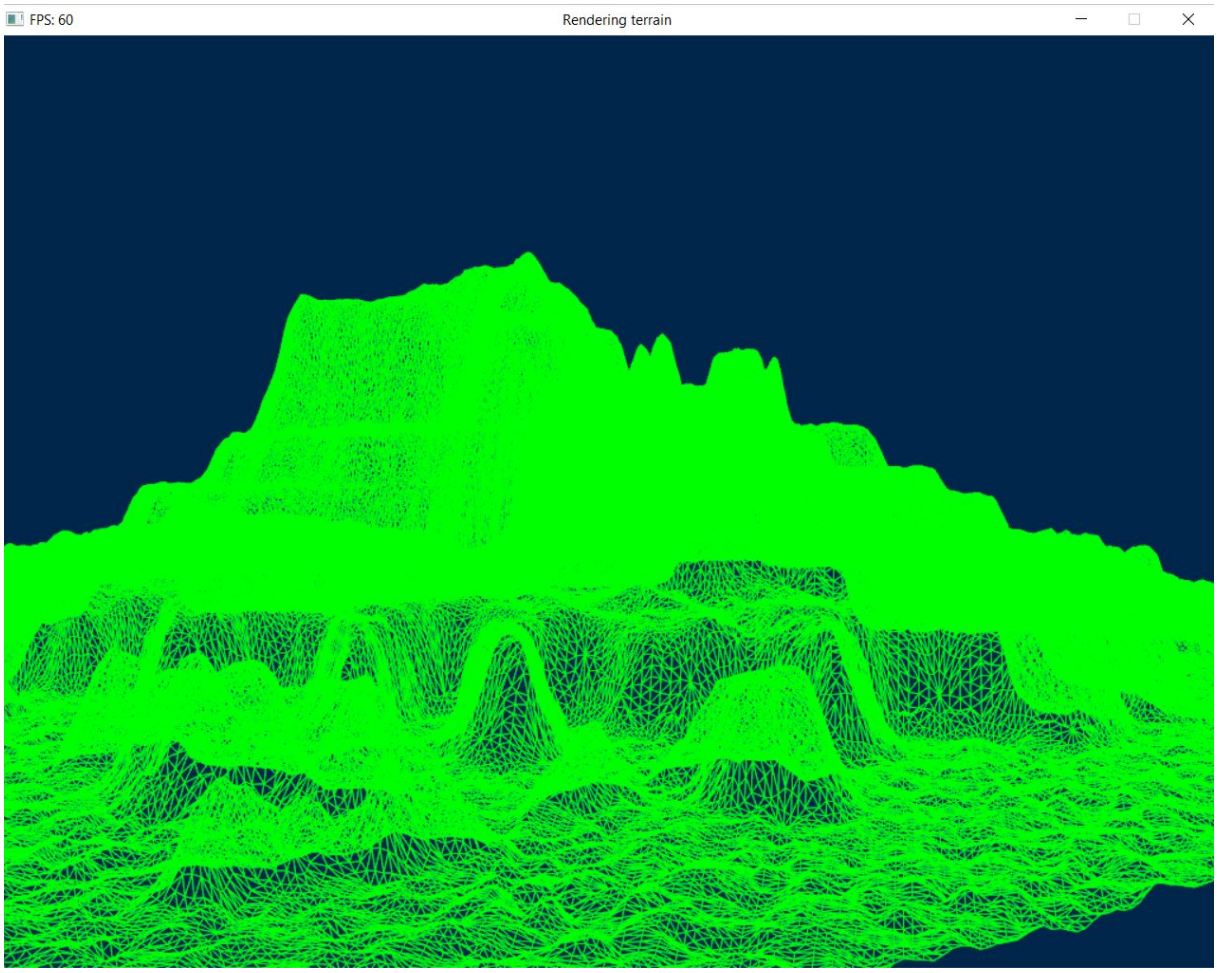


FIGURE 3.6 – Rendu de terrain utilisant la tessellation en mode filaire .

3.3.2 Placage de texture

La texture plaquée sur le terrain modélisé est générée par la méthode de génération de texture procédurale et voici un terrain aprés le placage de texture :

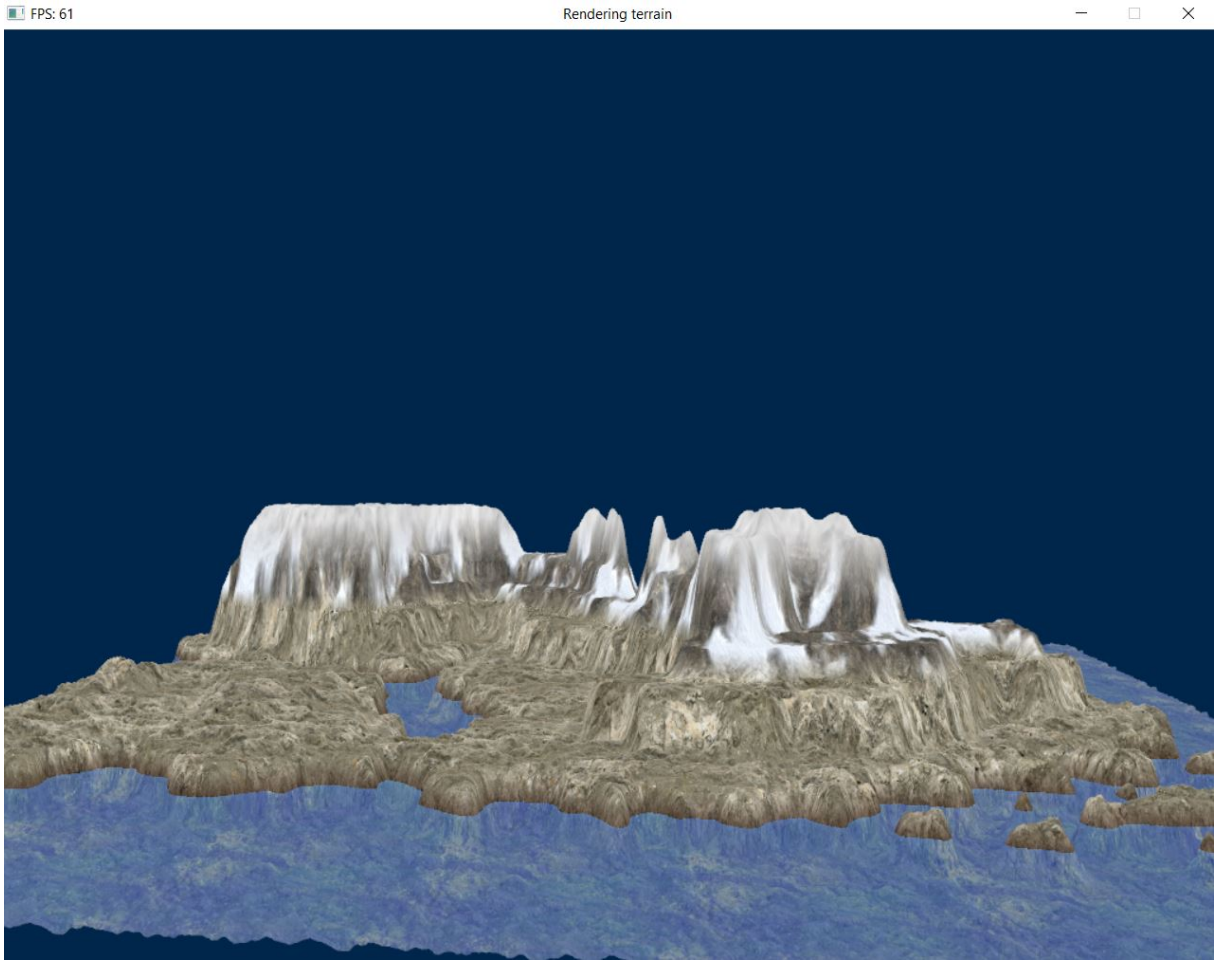
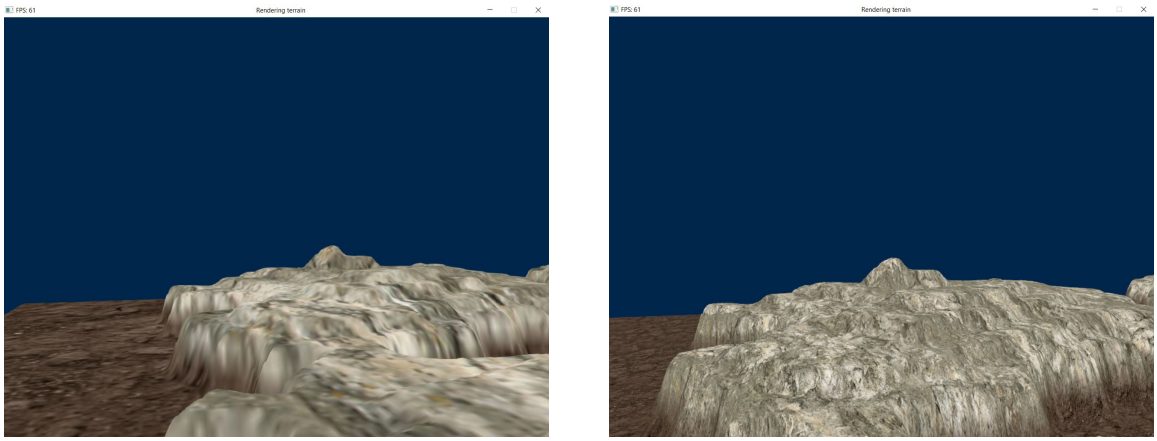


FIGURE 3.7 – Rendu de terrain texturée utilisant la tessellation.

3.3.2.1 Taille de la texture

Voici une comparaison entre une texture de taille 1024*1024 et une texture de taille 512*512. On voit parfaitement bien la différence de détail.

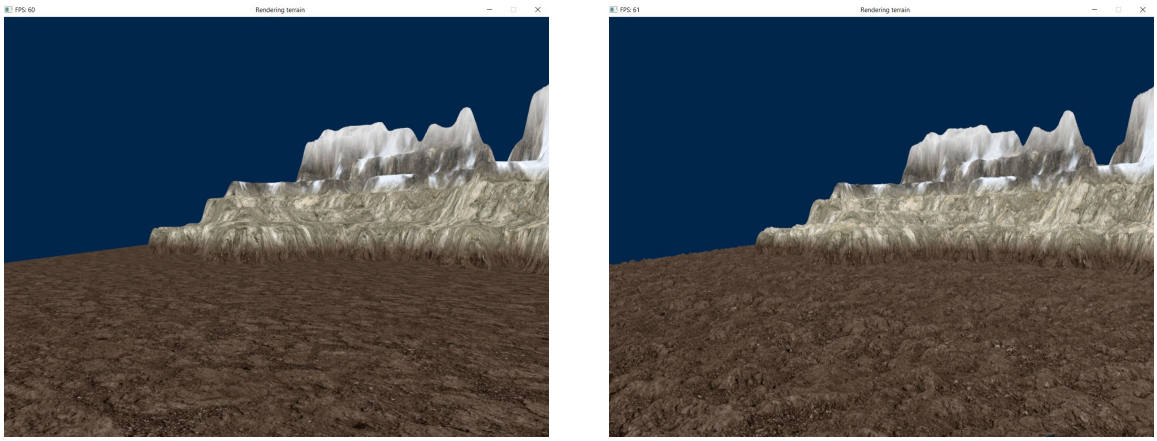


(a) (b)

FIGURE 3.8 – (a) texture de taille 512*512, (b) texture de taille 1024*1024

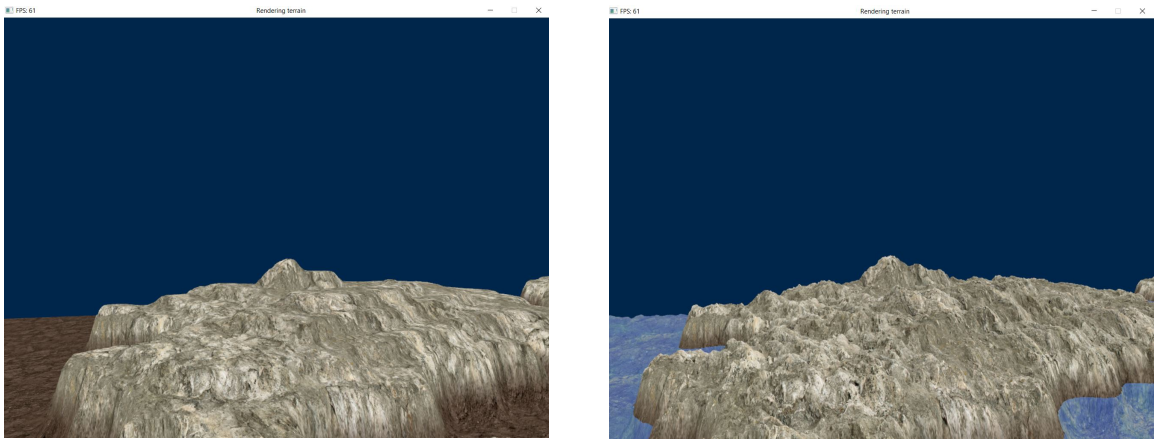
3.3.3 Textures détaillées

Nous avons comparer notre resultat avec et sans les textures détaillées.



(a) (b)

FIGURE 3.9 – (a) Sol sans les textures détaillées et (b) sol avec les textures détaillées



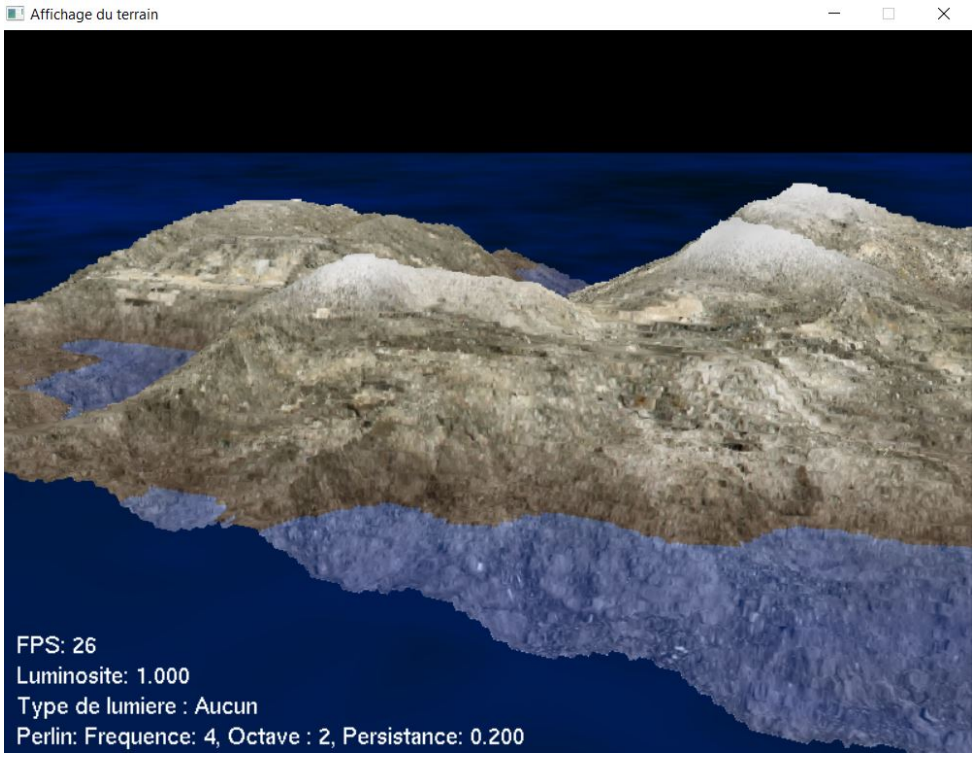
(a) (b)

FIGURE 3.10 – (a) Les roches sans les textures détaillées et (b) les roches avec les textures détaillées

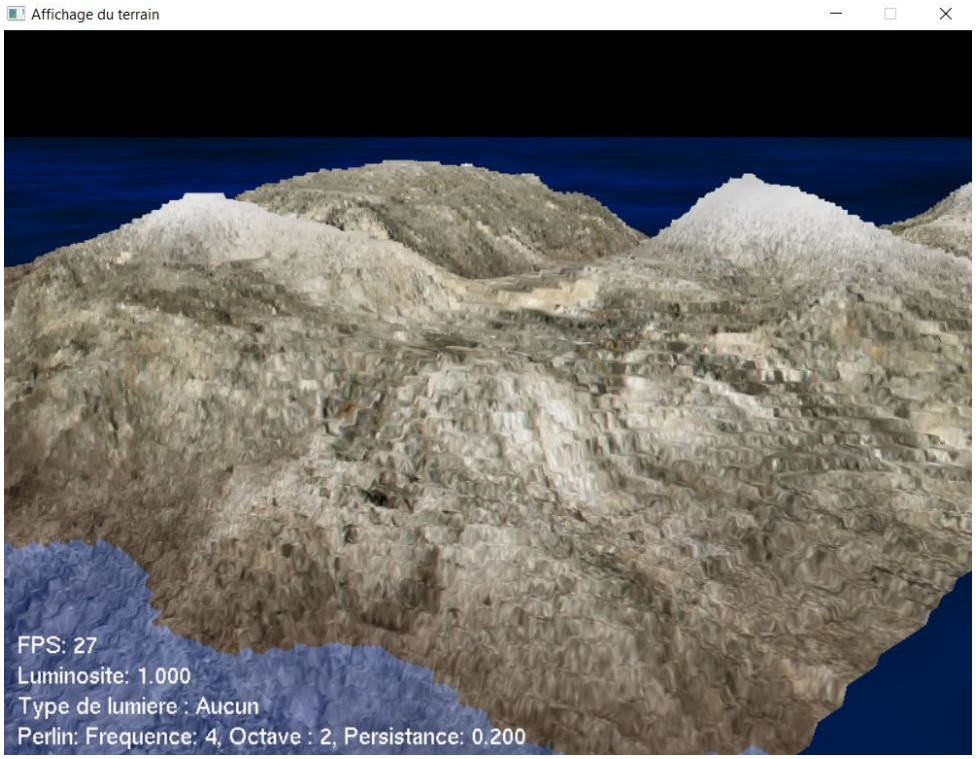
Maintenant, il ne reste plus qu'a comparer la qualité et la vitessee entre Perlin et notre application, et voici des terrains générés possèdent le même nombre de points :

Paramètres

- La fréquence de départ est 4.
- Le nombre d'octaves est 2.
- La persistance de départ est 0.2.

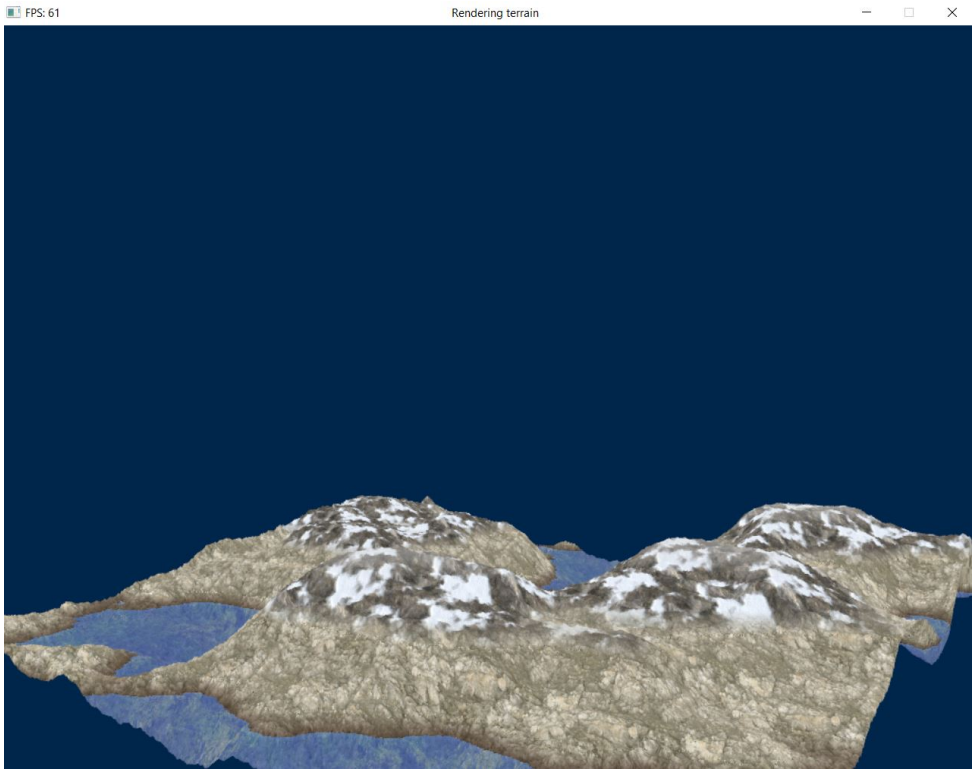


(a) FPS 26

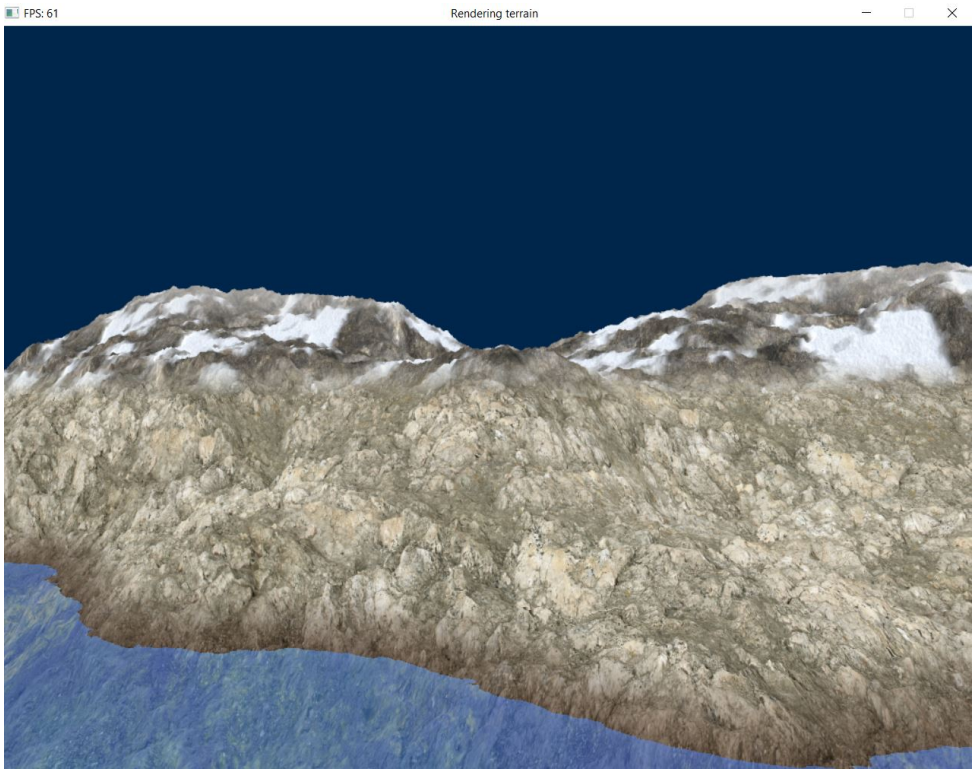


(b) FPS 27

FIGURE 3.11 – Résultats de Perlin



(a) FPS 61



(b) FPS 61

FIGURE 3.12 – Résultats de notre application en utilisant les GPU

FIGURE 3.13 – Résultat de notre application en utilisant une heightmap réalisé par logiciel de traitement d'image.

3.4 Conclusion

Dans ce chapitre nous avons présenté les résultats et la réalisation de notre application, nous avons présenté comment utiliser le GPU (les shaders) pour générer des terrains 3D convaincants et illimités à des fréquences d'images interactives. Nous avons également montré comment texturer le terrain pour l'affichage, construire un schéma de niveau de détail pour le terrain.

Les techniques que nous avons couvertes dans ce chapitre sont basées sur un nouveau paradigme pour la génération de terrain par GPU.

Conclusion générale

A travers ce travail nous avons présenté le rendu des terrains. Le rendu des terrains été un sujet de recherche actif en informatique graphique depuis de nombreuses années. Malgré les progrès considérables dans le domaine, ils restent de nombreux problèmes de contrôle de performances. L'accélération par le GPU est intéressante, la tesselationest une fonction puissante pour un rendu agréable.

En réalisant ce travail, nous avons pu acquérir :

- Des connaissances sur le rendu des terrains.
- Des connaissances sur l'exploitation des GPU et des caractéristiques matérielles et des API récentes.

Nous avons exploité ces connaissances pour effectuer le rendu des terrains à grande précision avec un nombre de sommet réduit ,ce qui réduit la bande passante et l'utilisation du mémoire grâce à la puissance de la fonction tessellation.

L'objectif global du travail est atteint. Nous pensons que ce travail peut être une brique de base pour des travaux plus affinés qui peuvent être traités et avoir réalisés.

Bibliographie

[1] V. Vanek, “Fractal geometry at secondary schools.”

[2] N. JANEY, “Les fractales, les I.F.S,” Mai 2018. [Online]. Available : http ://raphaello.univ-fcomte.fr/Ig/Fractales/Fractales.htm

[3] B. B. Mandelbrot, *The fractal geometry of nature*. WH freeman New York, 1983, vol. 173.

[4] A. Fournier, D. Fussell, and L. Carpenter, “Computer rendering of stochastic models,” in *Seminal graphics*. ACM, 1998, pp. 189–202.

[5] R. Krten, “Generating realistic terrain,” *Dr Dobb’s Journal-Software Tools for the Professional Programmer*, vol. 19, no. 7, pp. 26–31, 1994.

[6] M. A. DeLoura, *Game programming gems 2*. Cengage learning, 2001.

[7] H. Elias, “Spherical landscapes,” *disponível online em http ://freespace. virgin. net/hugo. elias/models/m_landsp. htm*, 2001.

[8] A. R. Fernandes, “The Fault Algorithm – Terrain Tutorial.” Mai 2018. [Online]. Available : http ://www.lighthouse3d.com/ opengl/terrain/index.php ?faultvar

[9] K. Perlin, “Implementing improved perlin noise,” *GPU Gems*, pp. 73–85, 2004.

[10] J. Bevins., “Libnoise library.” Mai 2018. [Online]. Available : http ://libnoise.sourceforge.net/coherentnoise/index.htm

[11] J. H. Clark, “Hierarchical geometric models for visible surface algorithms,” *Communications of the ACM*, vol. 19, no. 10, pp. 547–554, 1976.

[12] H. Hoppe, “Smooth view-dependent level-of-detail control and its application to terrain rendering,” in *Visualization’98. Proceedings*. IEEE, 1998, pp. 35–42.

[13] R. Pajarola, “Large scale terrain visualization using the restricted quadtree triangulation,” in *Visualization’98. Proceedings*. IEEE, 1998, pp. 19–26.

[14] R. Pajarola and E. Gobbetti, “Survey of semi-regular multiresolution models for interactive terrain rendering,” *The Visual Computer*, vol. 23, no. 8, pp. 583–605, 2007.

[15] B. Von Herzen and A. H. Barr, “Accurate triangulations of deformed, intersecting surfaces,” in *ACM SIGGRAPH Computer Graphics*, vol. 21, no. 4. ACM, 1987, pp. 103–110.

[16] M. Segal and K. Akeley, "The opengl," *Graphics System : A Specification (Version 4.0 (Core Profile)-March 11, 2010)• F S. Hill Computer Graphics Using Open GL*, 1992.

[17] P.-P. Sloan, "Normal mapping for precomputed radiance transfer," in *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. ACM, 2006, pp. 23–26.

[18] F. Policarpo and M. M. Oliveira, "Relief mapping of non-height-field surface details," in *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. ACM, 2006, pp. 55–62.

[19] N. Tatarchuk, "Dynamic parallax occlusion mapping with approximate soft shadows," in *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. ACM, 2006, pp. 63–69.

[20] W. Donnelly, "Per-pixel displacement mapping with distance functions," *GPU gems*, vol. 2, no. 22, p. 3, 2005.