République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Mohamed Khider

BISKRA

Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie

Département de l'informatique

N^o d'ordre :

Série :

Mémoire

En vue d'obtention du diplôme de Magister en informatique

Option : ***Synthèse d'images et vie artificielle***

illumination réaliste de terrains en temps réel

Réalisé par :

Mr. BELAICHE Hamza

Soutenu le : / /2010

Membres du jury :

Mr. DJEDI NourEddine	Président	Professeur, Université de Biskra
Mr. BABAHENINI Med Chaouki	Rapporteur	Maître de conférences A, Université de Biskra
Mr. CHERIF Foudil	Examinateur	Maître de conférences A, Université de Biskra
Mr. KHOLLADI Med KhirEddine	Examinateur	Maître de conférences A, Université de Constantine

Remerciements

Je tiens à remercier :

Mr. Med Chaouki BABAHENINI d'avoir accepté de m'encadrer et de gérer ce travail et pour ces précieux conseils.

Je remercie également les membres du jury :

Mr. DJEDI NourEddine	*Professeur, Université de Biskra*
Mr. CHERIF Foudil	*Maître de conférences A, Université de Biskra*
Mr. KHOLLADI Med KhirEddine	*Maître de conférences A, Université de Constantine*

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.

Dédicaces

A toi ma chère mère, pour tes encouragements, ton soutien inconditionnel et surtout ta présence à mes cotés dans les moments les plus difficiles, Que dieu te bénisse et te garde.

A mon cher père, pour ton soutien constant et inconditionnel, pour tes conseils et tes encouragements.

A mes chères frères, à mes chères sœurs et à toute ma famille.

A tous mes amis et collègues d'études.
A tous ceux qui m'ont aimé et me souhaitent le bonheur et la réussite.

Je vous dédie ce modeste travail

Belaiche Hamza

Résumé

L'illumination d'une scène est très importante pour son réalisme. Dans le cas d'un terrain cette illumination est assez complexe, puisqu'elle provient du soleil, du ciel, des ombres des nuages, des inter-reflections du terrain sur lui-même ou de l'interaction entre le terrain et les nuages. Le tout se déroulant dans un milieu qui diffuse plus ou moins la lumière dans toutes les directions : l'atmosphère.

Notre travail concerne l'étude du rendu de terrains, qui sont considérés comme des objets naturels. Néanmoins, la qualité visuelle qu'ils offrent est très souvent associée à une complexité dans la prise en charge des aspects modélisation.

Dans ce mémoire, nous avons fait une étude globale sur le domaine de la modélisation et du rendu de terrains. Nous avons ensuite proposé une approche de modélisation de terrain et le rendu associé, avec placage de texture et illumination temps réel.

Nos contributions essentielles se situent pour la modélisation dans hybridation des techniques diamond square avec celle de Perlin, et pour le rendu, nous avons proposé une technique de génération d'ombre. qui a permis d'améliorer considérablement le temps de calcul et la qualité visuelle.

Table des matières

Introduction générale **1**

I État de l'art 3

Introduction 4

1 Modélisation de terrain **6**

1.1 Introduction 6

1.2 Outils servant à la modélisation des terrains 7

1.2.1 Le concept fractales 7

1.2.2 Les cartes hauteurs et les terrains 9

1.3 Quelques algorithmes de génération de terrain 10

1.3.1 L'algorithme de subdivision en triangles 10

1.3.2 L'algorithme Diamond-Square 11

1.3.3 Modélisation des terrains par transformée de fourier 15

1.3.4 Technique de multiplication 17

1.3.5 Technique de Mid Point Displacement 17

1.3.6 L'algorithme Fault Formation 19

1.3.7 Le Circles Algorithm 19

1.3.8 La triangulation de Delaunay 20

1.3.9 La modélisaton des terrains par l'algorithme de Perlin 24

1.4 Bilan 35

1.5 Conclusion 38

2 Génération de textures de terrain **39**

2.1 Introduction 39

2.2 Utilisation de larges textures 39
2.3 Le mélange de textures de base 40
2.3.1 Calcul de la hauteur h 42
2.3.2 Calcul de la couleur du pixel 43
2.3.3 Taille de la texture 46
2.4 Le multi-texturing 47
2.5 Geo-mip-mapping et texture splatting 47
2.6 Mélange de texture de terrain par le materiel 49
2.7 Les détails textures 50
2.8 Conclusion 51

3 Illumination temps réel de terrain 52
3.1 Introduction 52
3.2 Présentation de l' l'illumination en plein air 52
3.3 Notations et terminologie de l'illumination 53
3.4 Calcul analytique temps réel de l'illumination en plein air 54
3.5 Méthodes de calcul d'ombres pour les terrains 60
3.5.1 Méthode de coefficient par différence entre les hauteurs 60
3.5.2 Limitations de la première méthode 62
3.5.3 Méthode de produit scalaire : L dot N 64
3.5.4 Méthode de lanceur de rayon 67
3.6 Bilan 71
3.7 Conclusion 72

II Contributions 73

4 Contributions 74
4.1 Introduction 74
4.2 Motivations pour de nouvelles approches 74
4.3 Approche proposée pour la modélisation 75
4.3.1 Principe 75
4.3.2 Amélioration de la technique proposée 81
4.3.3 Amélioration de l'approche proposée par la technique de Perlin 82

4.3.4 Le problème des deux techniques améliorées et sa résolution 82
4.4 Contributions pour l'illumination du terrain 83
4.4.1 Une première version de l'approche proposée 83
4.4.2 Version finale de l'approche proposée 88
4.5 Conclusion 93

5 Résultats et perspectives **94**
5.1 introduction 94
5.2 Génération de terrain 94
5.3 Placage de texture 95
5.4 Illumination 96
5.4.1 Résultats 96
5.4.2 Modélisation de lumière de la nuit 98
5.5 Modélisation 98
5.6 Exemple d'exploitation des résultats (simulateur de vol) 104
5.7 Perspectives 105
5.8 Conclusion 106

Conclusion générale **107**

Table des figures

1.1 Fractales déterministes 7
1.2 FBm avec des valeurs de H égales à 0.8, 0.6, 0.4, 0.3 et 0.2. 8
1.3 Un relief fractal généré par fBm représenté par facettes 9
1.4 Fractalisation d'un triangle 9
1.5 Technique de subdivision 10
1.6 Subdivision en triangles $r = 1.0$ 11
1.7 Subdivision en triangles $r = 1.5$ 11
1.8 Matrice initiale 12
1.9 La matrice après la phase du carré 12
1.10 La matrice après la phase du diamant 12
1.11 La matrice finale 12
1.12 Le carré 13
1.13 Le losange 14
1.14 Lissage à 0.1 15
1.15 Lissage à 0.5 15
1.16 Lissage à 0.9 15
1.17 DiamondSquare lisse 15
1.18 DiamondSquare rude 15
1.19 fourier $r = 2.5$ 16
1.20 fourier $r = 2.0$ 16
1.21 DiamondSquare^2 17
1.22 Fourier^2 $r = 1.7, 2.5$ 17
1.23 Technique de Mid Point Displacement en 3D 18
1.24 Itérations du Fault Algorithm 19
1.25 Itérations du Circles Algorithm 20

1.26 Les cercles circonscrits 21
1.27 Un côté légal 23
1.28 Un côté illégal (gauche), le flip (droite) 24
1.29 Terrain purement aléatoire 25
1.30 Même terrain dans le viewer de terrain de Fearyourself [STE96] 25
1.31 Interpolation bilinéaire 31
1.32 Interpolation linéaire 32
1.33 Interpolation non linéaire 32
1.34 Résultat incohérent 33
1.35 Résultat cohérent 34
1.36 Résultat lissé 35

2.1 Image de niveaux 40
2.2 Affichage de terrain 3D en filaire 40
2.3 Trois images de base 41
2.4 Texture générée à partir des trois images de base 41
2.5 Terrain texturé 41
2.6 Dégradé des images de base 43
2.7 Influence de la taille de la texture 47
2.8 Geo-mip-mapping 48
2.9 splatting 49

3.1 Angle horizon 55
3.2 horizon effectif 55
3.3 Visibilité du soleil 57
3.4 Comparaison de calculs d'ombre 60
3.5 Suivie de vecteur lumière 61
3.6 Défauts de la première méthode 62
3.7 Patch de taille égal à 2 63
3.8 Première méthode après l'application du filtre 64
3.9 Deuxième méthode (L dot N) utilisant un filtre 67
3.10 principe lanceur de rayon 68
3.11 Méthode du lanceur de rayon 70

4.1 Modèle mixte 80
4.2 Modèl Diamond-Square 81
4.3 Principe de cylindre 84
4.4 Résultat de la première version 87
4.5 Problème de la première solution 88
4.6 Principe de cône 89
4.7 Solution de la version finale 92
4.8 solution de la méthode de lanceur de rayon 92

5.1 Terrain de perlin en mode filaire 95
5.2 Terrain de perlin texturée 95
5.3 Résultats de l'illumination sans utiliser le filtre 96
5.4 Résultats de l'illumination avec l'utilisation du filtre 96
5.5 Illumination sans utiliser le filtre 97
5.6 illumination avec l'utilisation du filtre 97
5.7 Lumière de nuit 98
5.8 Modèle mixte 99
5.9 Modèle Diamond-Square 99
5.10 Modèle mixte à texture de faible détail 100
5.11 Modèle Diamond-Square à texture de faible détail 100
5.12 Résultats de Perlin 101
5.13 Résultats de Diamond-Square 102
5.14 Résultats du modèle mixte 103
5.15 Principe de simulateur de vol 105

Introduction générale

Il y a déjà 35 ans que les premiers dessins par ordinateur ont été produits notamment pour le système de défense et de contrôle aérien SAGE et au M.I.T. avec l'ordinateur TX1. Pendant plus de 25 ans, certains scientifiques ont utilisé la capacité des ordinateurs pour produire des diagrammes et des graphes dans leurs rapports, thèses et articles. Pendant toute cette période, le public, même le public averti tel que la communauté universitaire, ignorait littéralement les possibilités graphiques de l'ordinateur. Il faut tout de même convenir que le matériel était encore cher, pas toujours très maniable et les résultats pas toujours très spectaculaires [DAN03].

Aujourd'hui, la situation a radicalement changé, tous les micro-ordinateurs personnels ont des capacités graphiques. Les millions de téléspectateurs des pays occidentaux sont envahis par les génériques et logos produits par ordinateur. On est d'ailleurs parfois ébahi par le réalisme et la perfection de certaines images générées par ordinateur. Que ce soit dans le domaine technique, médical ou simplement artistique, ces images forcent notre admiration et nous questionnent. Dans le domaine de la synthèse d'images réalistes, en moins de dix ans, on a assisté à des progrès spectaculaires, **autant du point de vue matériel que du point de vue logiciel**. Il faut d'ailleurs remarquer que leur évolution est intimement liée. Il serait difficilement concevable de produire des images réalistes sans disposer d'un terminal graphique de haute résolution avec au moins quelques centaines de couleurs affichables simultanément [DAN03].

Plus récemment, c'est le domaine du multimédia qui s'est développé grâce aux progrès fulgurant dans les télécommunications. Avec Internet et World Wide Web, on peut consulter ou échanger des images créées à l'autre bout du monde. Par la réalité virtuelle, on peut plonger aussi dans des mondes virtuels et de nouveau, grâce aux télécommunications rapides comme les réseaux ATM, il est même possible de partager les mondes virtuels entre des participants du monde entier [DAN03].

Le progrès spectaculaires de matériel ne suffit pas il faut aussi un progrès logiciel. La conception d'une application graphique à affichage 3D suit plusieurs étapes :

- détermination des scènes devant être représentées,
- modélisation géométrique de ces scènes,
- placement des textures et des lumières,
- programmation.

Pour avoir ce progrès logiciel, il faut apporter des optimisations dans les algorithmes utilisés dans chaque étape de conception. L'optimisation généralement réduit l'efficacité, alors il faut chercher des compromis.

Objectif : Notre travail concerne l'étude du rendu de terrains qui sont considérés comme des objets naturels, cette problématique intéresse de plus en plus les laboratoires de recherche en synthèse d'images. Néanmoins, la qualité visuelle qu'ils offrent est très souvent associée à une complexité dans la prise en charge des aspects modélisation.
Notre objectif principal est de proposer des algorithmes optimisés et efficaces (temps réel) et d'aboutir à un aspect simulation la plus naturelle possible.

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

- **Chapitre 1 " Modélisation de terrain "** : Dans ce chapitre, nous allons examiner quelques algorithmes pour la génération des terrains, ces algorithmes permettent de produire un modèle numérique pour notre terrain.
- **Chapitre 2 " Génération de textures de terrain "** : Dans ce chapitre nous allons exposer un ensemble de techniques d'habillage de terrain, qui consiste à colorer le terrain en espérant obtenir un résultat suffisamment réaliste.
- **Chapitre 3 " Illumination temps réel de terrain "** : Dans ce chapitre nous allons présenter quelques techniques d'illumination et de calcul des ombres pour le terrain, pour bien montrer le relief, et les effets des ombres.
- **Chapitre 4 " Fondements de la méthode "** : dans ce chapitre, nous allons essayer d'introduire de nouvelles méthodes pour l'illumination et la modélisation qui répond aux critères (obtimalité, efficacité).
- **Chapitre 5 " Résultats et perspectives "** : Ce chapitre est consacré à la présentation des résultats permettant d'évaluer les perfermences des nouvelles méthodes proposées.

Ces cinq chapitres seront suivis par une conclusion qui présente le bilan de ce travail et propose ses perspectives potentielles.

Première partie

État de l'art

Introduction

Les terrains jouent un rôle fondamental dans la création d'une scène naturelle. Les techniques de génération automatique ont de nombreuses applications comme les simulateurs de vol, les effets spéciaux au cinéma ou les jeux vidéos. L'augmentation de la puissance de calcul combinée à des méthodes de visualisation de plus en plus performantes ont permis de créer des terrains réalistes.

Trois techniques de génération de terrains existent : les modèles fractals, les méthodes de simulation d'érosion et les algorithmes de synthèse à partir d'exemples.

La synthèse procédurale consiste à générer un terrain automatiquement par des constructions géométriques ou des combinaisons de fonctions de bruit [MAN82, MKM89, ST89, JS06], un état de l'art est présenté dans [EMP*98]. Ces méthodes produisent de très grands terrains qui manquent parfois de réalisme. Plusieurs problèmes ressortent de ces méthodes. L'utilisation de méthodes automatiques ne permet pas de contrôler la forme finale du terrain. L'utilisation de cartes d'élévation [AND03] de données limite fortement la topologie et la géométrie du terrain. Gamito et Musgrave [GM01] améliorent les cartes d'élévation à l'aide de courbes de profills pour générer procéduralement des terrains avec des surplombs. Leur approche ne permet cependant pas la création de toutes les formes volumiques comme les grottes ou les arches [AEJS08].

Les méthodes de simulation d'érosion permettent de générer des terrains physiquement plausibles [MKM89, NWD05, KMN88, RPP93, CMF98]. Ces méthodes s'appuient sur des simulations plus ou moins précises pour déterminer le transport et le dépôt de matière sur des cartes d'élévation de données. Plusieurs méthodes [NWD05, MDH07] ont été portées sur GPU pour accélérer les temps de calcul coûteux de la simulation. Pour simuler l'érosion sur des terrains tridimensionnels, Ito [IFMC03] et Beardall [BFO07] utilisent des grilles de voxels mais la taille du terrain reste limitée sur à cause du coût de la structure de données. L'utilisation d'au plus deux matériaux (la roche et les sédiments) pour simuler l'érosion est une autre limitation de ces approches [AEJS08].

Pour contrôler la génération du terrain, Zhou [ZSTR07] a mis en place une procédure s'appuyant sur la synthèse de textures. Cette méthode permet de contrôler globalement la forme finale du terrain, mais s'appuie également sur une représentation à base de cartes d'élévation de données.

Ces méthodes ont deux principales limitations. Elles utilisent essentiellement des cartes d'élévation de données limitant la topologie et la géométrie du terrain en ne permettant pas

de modéliser des surplombs, des falaises, des arches ou des grottes. D'autre part, le terrain est souvent considéré comme un unique matériau homogène ce qui diminue le réalisme et les possibilités de modélisation.

Dans notre étude on s'intéresse aux modèles fractals ou (modèles à synthèse procédurale) qui permet la génération automatique des terrains.

Chapitre 1

Modélisation de terrain

1.1 Introduction

Il n'est pas surprenant que les objets les plus faciles à construire par ordinateur soient justement ceux que l'humain a inventés : les maisons, les voitures, les avions. Par contre, les objets de la nature et les phénomènes naturels en généralement donne des casses têtes aux chercheurs. Les scènes naturelles offrent un domaine d'étude extrêmement vaste et complexe, ne serait- ce par la diversité des formes y prenant place : plantes, terrains, fourrures, nuages, feu, fluides, etc. Un des objectifs premiers de la synthèse d'images étant de reproduire l'apparence de la réalité, on dispose ici d'un vaste terrain de jeu où quelques jalons ont été posés, mais où il reste encore beaucoup à faire.

On distingue souvent deux aspects dans la complexité d'une scène naturelle : les objets naturels et les phénomènes naturels. Les objets naturels sont concrets et solides, ils interagissent avec la lumière par des modèles matière lumière, ce sont par exemple les terrains, les végétaux, etc. Les phénomènes naturels ne sont pas tangibles, leur surface n'est pas clairement définie, ce sont par exemple le brouillard, le feu, le vent, etc. Si les objets naturels sont tangibles, les mécanismes entrant en jeu dans leur modélisation et leur apparence sont extrêmement complexes. Par exemple, la richesse en eau du terrain, la quantité de lumière reçue, etc.

La génération de terrain est souvent un sujet qui intéresse et passionne. Lorsqu'on affiche un terrain en 3 dimensions, il faut arriver à colorer ou utiliser une texture pour le rendu des triangles. Pour un rendu plus réel, la texture doit être appliquée correctement. Dans ce chapitre, nous allons essayer d'entamer quelques techniques de modélisation des terrains.

L'importance de la modélisation des terrains ouvre une grande porte vers la recherche

des techniques pour générer les terrains. Les paragraphes ci dessous sont consacrés à décrire quelques techniques pour avoir une vision générale concernant la génération de terrain. ces mécanismes sont basés beaucoup plus sur la notion de fractales.

1.2 Outils servant à la modélisation des terrains

1.2.1 Le concept fractales

Principe et théorie Une approche intuitive consiste à considérer qu'une fractale [BEN77] est la transformation récursive d'un objet par n copies de lui même à l'échelle r (avec $r < 1$). Cette propriété est appelée l'autosimilarité. 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".

Fractales déterministes On appelle fractale déterministe une fractale dont le mode de réplication ne fait pas intervenir de composante aléatoire.

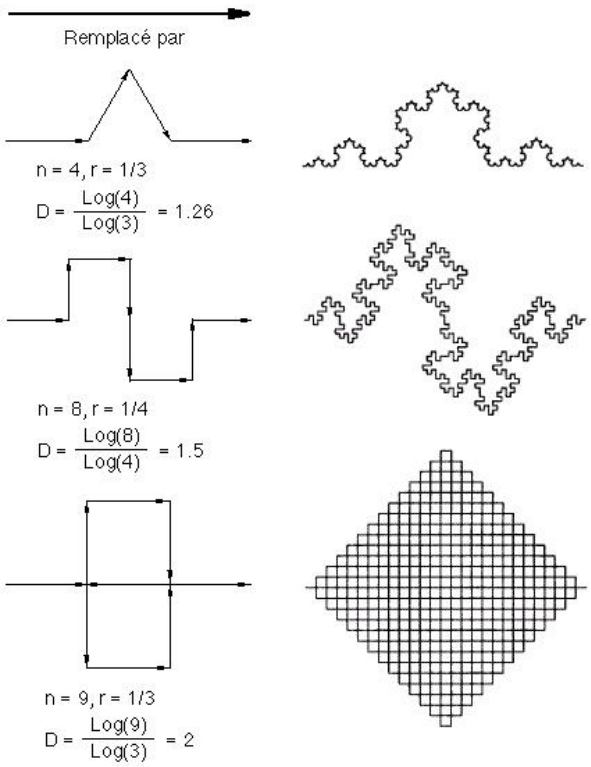


FIG. 1.1 – Fractales déterministes

Fractales non déterministes Les fractales stochastiques, par opposition aux fractales déterministes, permettent de générer des dessins non réguliers. Pour générer une fractale stochastique, on n'a plus un seul mode de réplication, mais deux ou plusieurs choisis aléatoirement.
On obtient de bons résultats avec D voisin de 1,2 pour générer des rivages océaniques (lignes) et avec D voisin de 2,2 pour la génération de montagnes (surfaces).
Plus D croît plus la ligne ou la surface devient chaotique et irréaliste.

Historiquement les fractales sont issues des travaux de Mandelbrot et Van Ness sur le "mouvement Brownien fractionnaire" (fractional Brownian motion, fBm).
Il s'agit de caractérisation une famille de processus stochastiques Gaussiens unidimensionnels VH(t) (H : réel compris entre 0 et 1, t : le temps).
Plus H est proche de 0 plus la courbe est bruitée. Une valeur de H égale à 0,5 quantifie une mouvement Brownien normal.

Autoaffinité Le principe de l'autoaffinité correspond aux faits suivants :

1. Deux morceaux quelconques de la courbe représentative se ressembleront,
2. Si on effectue une mise à l'échelle d'un facteur r sur le temps et d'un facteur rH sur VH(t), la courbe garde toujours le même aspect.

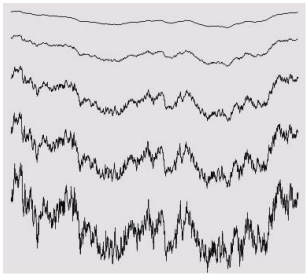


FIG. 1.2 – FBm avec des valeurs de H égales à 0.8, 0.6, 0.4, 0.3 et 0.2.

L'extension multidimensionnelle des fBm permet de modéliser un large éventail de phénomènes naturels.
Pour les reliefs, le paramètre temps t est transformé en un couple (x,y) indiquant une position dans le plan, VH(x,y) donne l'altitude du point P.

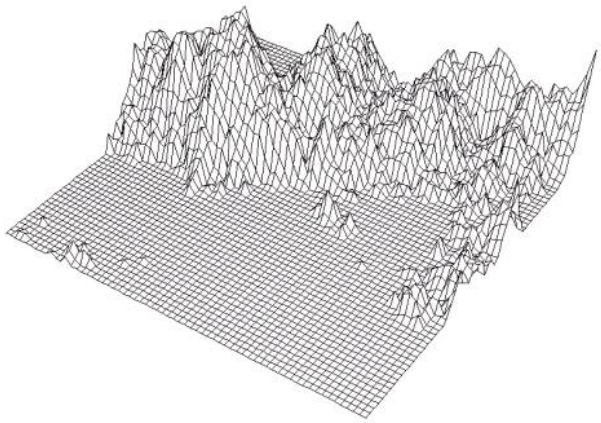


FIG. 1.3 – Un relief fractal généré par fBm représenté par facettes

Un objet fractal de dimension fractale D apparaît comme un fBm de coefficient H tel que D = E + 1 - H où E est le nombre de variables de la fonction fBm, c'est à dire la dimension euclidienne de l'objet construit. Cet objet n'est évidemment pas strictement autosimilaire mais seulement autoaffine.

Implantation des fractales stochastiques La programmation exacte des lois mathématiques requiert une quantité de calcul très importante. Les implantations sont essentiellement des approximations du mouvement Brownien fractionnaire
Soit par des techniques spatiales (subdivision récursive),

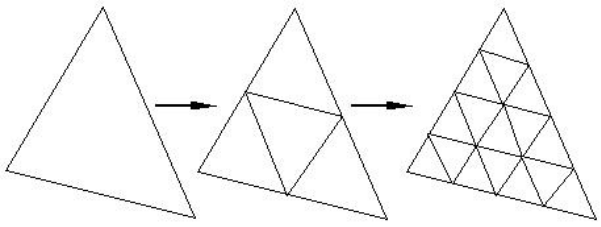


FIG. 1.4 – Fractalisation d'un triangle

Soit par des techniques spectrales (par filtrage de Fourier).

1.2.2 Les cartes hauteurs et les terrains

La représentation la plus utilisée pour modéliser un terrain est la carte d'élévation[AND03]. Avec cette représentation, un terrain est stocké sous forme d'un tableau à deux dimensions, où chaque case contient la hauteur du terrain. De plus on peut associer une couleur à chaque case,

ce qui revient à associer une texture au terrain. La souplesse de cette représentation provient de la possibilité de considérer ces cartes comme des images 2D dont la manipulation est très intuitive :

- modélisation possible avec des outils de dessin 2D,
- facilité de génération de niveaux de détails,...etc.

1.3 Quelques algorithmes de génération de terrain

1.3.1 L'algorithme de subdivision en triangles

L'algorithme de subdivision en triangles [FB91] prend un terrain triangle, et dans chaque itération on subdivise chaque triangle en quatre triangles plus petit, puis on applique un facteur de perturbation a chaque nouveau sommet créé. La procédure de génération est la suivante :

1. Commencer par un triangle. Choisir une altitude aléatoire pour les points finaux.
2. Pour chaque arête de triangle. On détermine le point milieu et on applique une perturbation aléatoire. La perturbation est distribuée entre $-k^r$ et k^r , où k est la longueur de l'arrête subdivisée et r est un paramètre de rudesse.
3. Diviser le triangle en quatre plus petit triangles par la connexion des points milieu, répéter le processus pour chaque petit triangle.

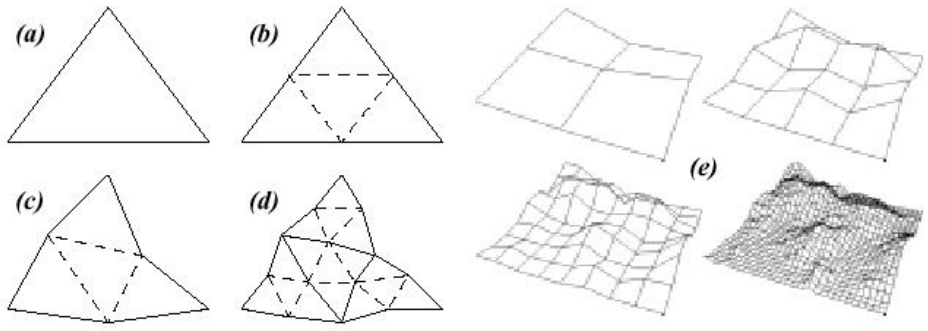


FIG. 1.5 – Technique de subdivision

On répète le processus jusqu'au niveau de détail désiré. A chaque itération la perturbation aléatoire se réduit car les arêtes serons plus petit.
Cet algorithme possède un paramètre rudesse qui contrôle le terrain résultat. avec un grand r la perturbation décroître plus rapidement après chaque itération qu'avec un petit r.

Alors avec un grand r, la perturbation devient petit dans le haut niveau de détail. Le résultat est un terrain lisse. Contrairement avec un petit r le résultat est un terrain rude pour cela r est appelé paramètre de rudesse.
L'algorithme peut être appliqué à une grille carrée par division au diagonal en deux triangles.

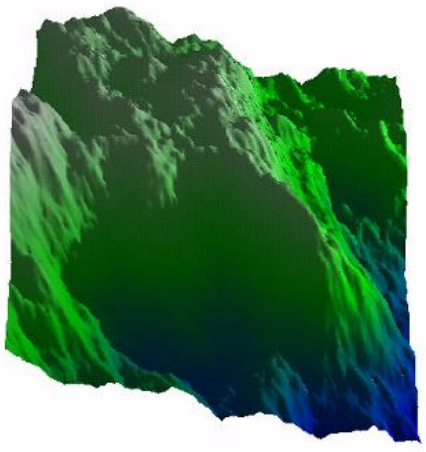


FIG. 1.6 – Subdivision en triangles $r = 1.0$

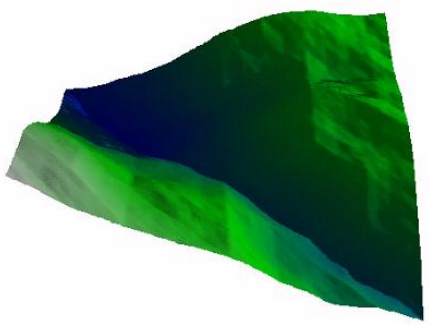


FIG. 1.7 – Subdivision en triangles $r = 1.5$

Un défaut clair de cet algorithme est l'apparence des rainures entre les arêtes des triangles. Ceci est plus clair quand le terrain est plus lisse, comme dans la figure précédente il y a une rainure qui forme une diagonale principale. Ce résultat est dû à la géométrie de l'algorithme. La raison pour la quelle cette rainure apparaît est que l'altitude de chaque point se détermine en fonction de deux points seulement et les autres points ne sont pas pris en considération [FB91].

1.3.2 L'algorithme Diamond-Square

L'algorithme DiamondSquare [FB91, GJE84], comme son nom l'indique, se base sur 2 phases : la phase du carré et la phase du diamant. Il s'agit d'itérer ces 2 phases jusqu'à calculer tous les points de la matrice représentant le terrain 3D.
En effet, la largeur de la matrice représente les abscisses, la longueur correspond aux ordonnées et enfin chaque "case" de la matrice représente la hauteur relative du point dont on vient de décrire les 3 composantes pour le situer dans un repère : pour accéder à une case de la matrice on fait $tab[x][y] = z$; ce qui indique que les 3 coordonnées sont présentes.
Pour mieux comprendre le fonctionnement, prenons un exemple simple pour expliquer l'algorithme. Soit une matrice 5X5 :

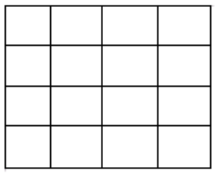


FIG. 1.8 – Matrice initiale

La phase du carré consiste à élever le centre du carré d'une valeur aléatoire (le point en rouge) :

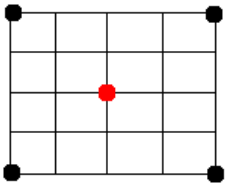


FIG. 1.9 – La matrice après la phase du carré

La phase du diamant consiste à élever le centre de chaque losange formé par le centre et les coins du carré précédent d'une valeur aléatoire (les points en bleu). On peut constater déjà qu'on se retrouve confronté à un problème : il n'y a pas quatre points complet pour former le diamant. Il faut donc tenir compte du cas particulier des bords :

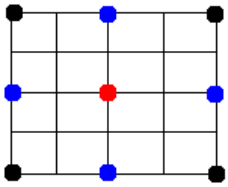


FIG. 1.10 – La matrice après la phase du diamant

On réitère avec les carrés obtenus (vert), puis les diamants (orange) et ainsi de suite jusqu'à parcourir l'ensemble de la matrice :

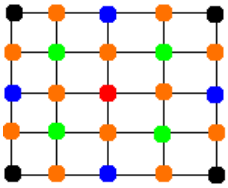


FIG. 1.11 – La matrice finale

On obtient donc l'algorithme suivant :

→ $espace = Largeur$;
→ $tant\ que\ espace > 1\ faire$
→ {
→ $espace = espace/2$;
→ $pour\ chaque\ carre\ de\ largeur\ espace\ faire$
→ {
→ $etape\ du\ carre$;
→ }
→ $pour\ chaque\ diamant\ de\ largeur\ espace\ faire$
→ {
→ $etape\ du\ diamant$;
→ }
→ }

La phase du carré calcule la moyenne des 4 points formant le carré. Lors de cette phase, on est positionné sur le centre ayant pour coordonnées (x, y). Sur un carré (a, b, c, d), on retrouve les coordonnées suivantes :

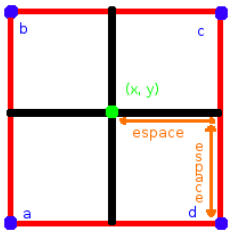


FIG. 1.12 – Le carré

$a = (x - espace, y - espace)$
$b = (x - espace, y + espace)$
$c = (x + espace, y + espace)$
$d = (x + espace, y - espace)$

Mais il faut faire attention aux limites de la matrice. Il faut tester les coins du carré pour que leurs coordonnées soient comprises entre 0 et $(largeurMatrice - 1)$.

La phase du diamant calcule la moyenne des 4 points formant le losange (représentant le diamant). on est positionné sur le centre qui a pour coordonnées : (x, y). Sur un diamant (a, b, c, d),

on retrouve les coordonnées suivantes :

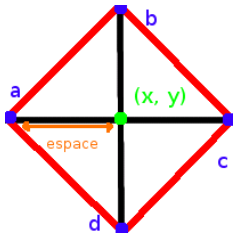


FIG. 1.13 – Le losange

$a = (x - espace, y)$

$b = (x, y + espace)$

$c = (x + espace, y)$

$d = (x, y - espace)$

Il faut encore faire attention aux limites de la matrice !

Comme le terrain est aléatoire, la hauteur d'un point doit être choisit au hasard. Néanmoins, pour ne pas avoir n'importe quoi, il faut quand même contrôler sa valeur. On parle alors de hauteur pseudo aléatoire car cela n'est pas totalement dû au hasard.

Tout d'abord, la hauteur d'un point est calculée en fonction de la moyenne de la hauteur des points que l'algorithme a utilisé pour l'obtenir.

On y ajoute une valeur aléatoire , à laquelle on applique un coefficient de lissage pour ne pas avoir un terrain en dents de scie. Ensuite, on multiplie à cette valeur alétoire, l'espace entre le point cible et les points sources, afin de garder la cohérence du terrain. Voici donc la formule pour obtenir la hauteur :

$hauteur = moyenne() + random() * espace * lissage$

Le coefficient de lissage est fractionnaire et donc inférieur à 1. Plus la valeur est élevée, moins le terrain sera lisse :

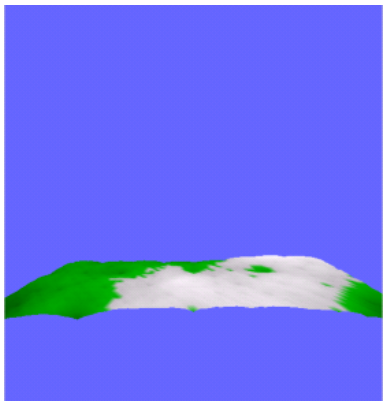


FIG. 1.14 – Lissage à 0.1

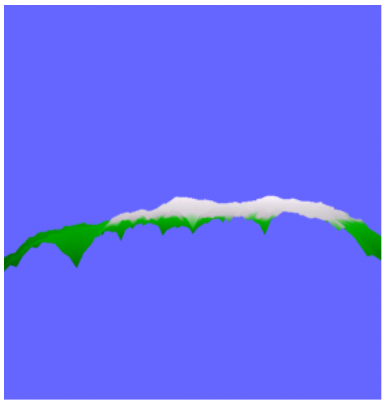


FIG. 1.15 – Lissage à 0.5

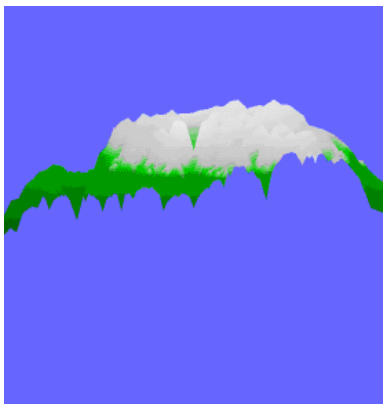


FIG. 1.16 – Lissage à 0.9

Les figures suivantes représentent deux terrains générés par l'algorithme du DiamondSquare :

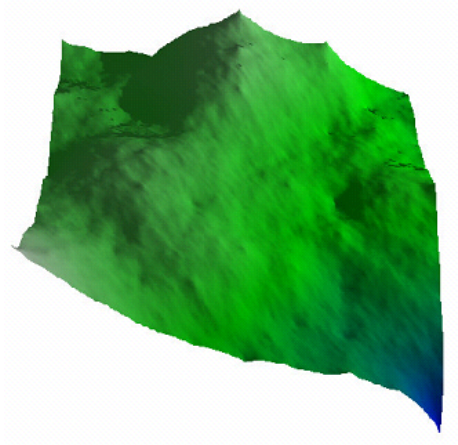


FIG. 1.17 – DiamondSquare lisse

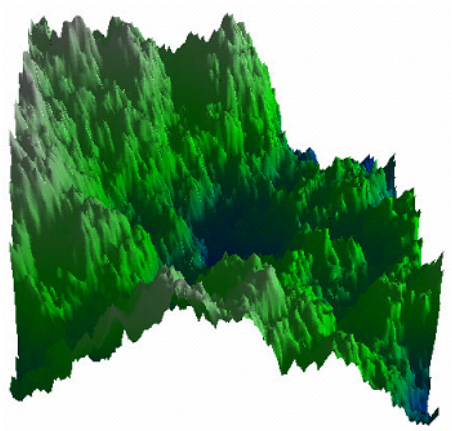


FIG. 1.18 – DiamondSquare rude

Le défaut posé par l'algorithme décomposition en triangles ; l'apparence des rainures entre les arêtes des triangles, a été réglé ici par la dépendance de chaque point des autres points dans les quatre directions, mais pour un terrain lisse on aura le problème d'apparition des petites crêtes (sommets) [FB91]. Cet algorithme est mieux que le précédent mais le terrain obtenu n'est pas vraiment aléatoire.

1.3.3 Modélisation des terrains par transformée de fourier

La transformée de fourier [FB91] est basée sur l'idée qu'une fonction peut être représentée sous la forme d'une somme de sinus et de cosinus dans les différentes fréquences. La transformée de fourier représente une fonction sous cette forme dans le domaine fréquentiel.

La transformée de fourier discrète (DFT) est une transformée de fourier appliquée a un ensemble de valeurs discrètes. Le résultat est un ensemble de nombres complexes qui possède

la même taille que l'ensemble original, les valeurs en résultat peuvent être considérées comme des amplitudes pour les différentes fréquences de l'ensemble original. Le processus peut être inversé sans aucun changement de l'ensemble de départ. La DFT peut être améliorée par $O(n - \log(n))$ en utilisant la transformée de fourier rapide (FFT), où n est la taille de l'ensemble de valeurs.

L'idée de la transformée de fourier est de considérer le terrain comme un ensemble d'ondulation à fréquences différentes. L'amplitude de ces ondulations décroît avec la fréquence. C'est la même chose comme l'algorithme précédent ; la perturbation décroître avec le degrie d'itération.

L'algorithme de la transformée de fourier est différent par rapport aux algorithmes précédents, car il n'est pas un processus itératif. Il peut être résumé par les étape suivante :

1. Créer une grille 2-dimensionnelle de valeurs aléatoires,
2. Appliquer la FFT 2-dimensionnelle,
3. Ajuster chaque valeur dans la transformée par $\frac{1}{f^r}$, où f est la fréquence représentée par sa valeur et r le paramètre de rudesse,
4. Appliquer la FFT inverse. Le résultat est un terrain fractal.

Le terrain résultat ne contient ni rainures artificielles ni crêtes, a l'inverse des algorithmes précédents. Ilpossède la propriété qu'un coté est similaire au coté en face.

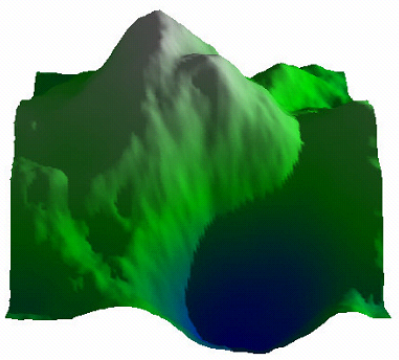


FIG. 1.19 – fourier $r = 2.5$

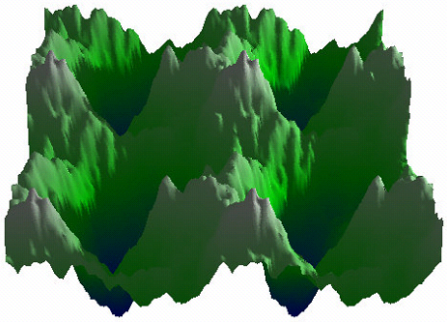


FIG. 1.20 – fourier $r = 2.0$

1.3.4 Technique de multiplication

A ce stade les terrains générés par les algorithmes précédents ne sont pas réalistes. Les terrains générés contiennent des crêtes et des vallées qui possèdent les mêmes caractéristiques.

Dans le monde réel les vallées sont plus lisses que les crêtes, ceci est le résultat de l'érosion qui rend les vallées plus lisses.

La solution est de simuler le processus de l'érosion, par la modification de l'algorithme de DiamondSquare ou subdivision en triangles pour résoudre le problème des vallées lisses et crêtes rudes.

Une solution rapide et simple consiste à multiplier deux terrains entre eux. Pour cela il suffit de générer deux terrains avec des altitudes entre 1 et 0, puis multiplier ces altitudes pour créer le nouveau terrain.
Toutes les altitudes se réduisent mais les plus basses plus que les plus hautes. Le terrain résultant est conforme, à ce qu'on recherche, étant donnée qu'il contient des vallées lisses et des crêtes rudes.

Les deux figures suivantes représentent deux terrains générées par la technique de multiplication [FB91] la première par la méthode de DiamondSquare et la deuxième par la transformée de fourier.

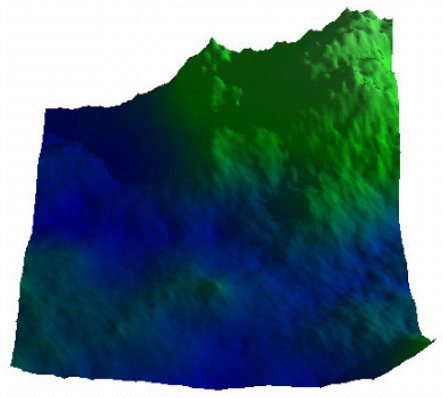


FIG. 1.21 – DiamondSquare[2]

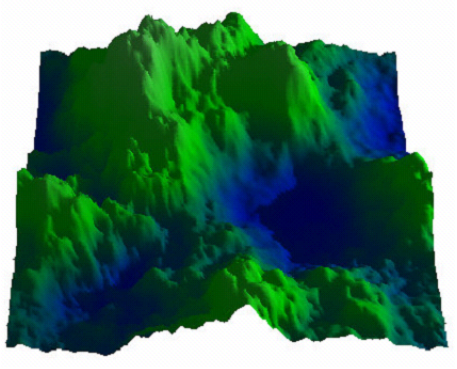


FIG. 1.22 – Fourier[2] $r = 1.7, 2.5$

1.3.5 Technique de Mid Point Displacement

Le Mid Point Displacement [AND03] ou MPD est basé sur les fractales, il fonctionne donc de manière récursive. Pour mieux comprendre son fonctionnement on va étudier le cas 2D.

Le principe est donc de prendre un segment, de trouver son milieu et de lui appliquer une transformation aléatoire.

Segment

Transformation du milieu

Une fois cette étape achevée, on réitère le processus sur les 2 segments de part et d'autre du milieu, et ainsi de suite.

Transformation des milieux de chaque segment

Passons maintenant à généralisation dans le cas 3D, au lieu de travailler sur un segment on va travailler sur un carré, notre heightmap, dont chaque cellule est initialisé à 0, on revient alors à la technique de subdivision récurssive qui travail sur les carrés au lieu des triangles.

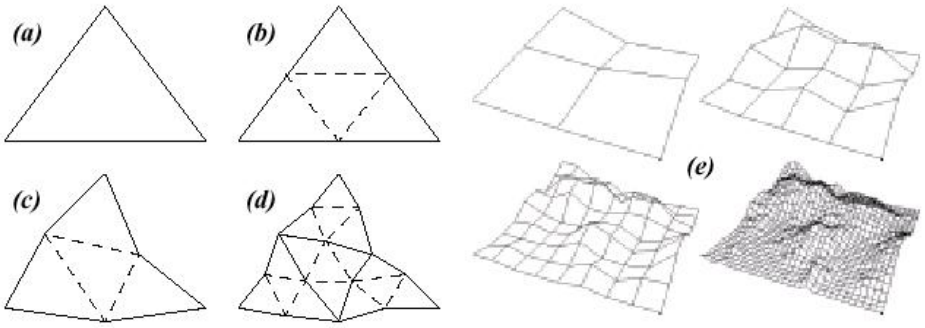


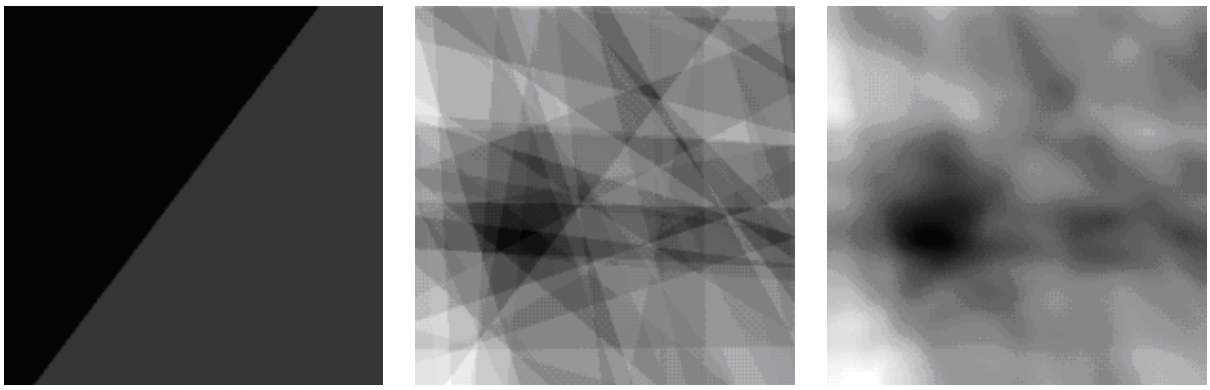
FIG. 1.23 – Technique de Mid Point Displacement en 3D

Avantage : Le principal avantage de cette méthode est le fait que l'on est certain du paysage que l'on va représenter, étant donné qu'il est figé sur fichier. On peut aussi le retoucher afin de l'améliorer ou alors, on a la possibilité d'ajouter facilement des décors selon nos envies[AND03].

Inconvénient : Le principal inconvénient est que le terrain généré consomme beaucoup d'espace mémoire pour être stocké, un terrain d'une taille de 1024×1024, avec les altitudes stockés sur 16 bits, prend effectivement 2 Méga octets, sans compter les informations complémentaires que l'on désire y ajouter[AND03].

1.3.6 L'algorithme Fault Formation

L'Algorithme Fault Formation [AND03] fonctionne de manière très simple. Il trace une ligne sur notre heightmap, tous les points d'un côté de la ligne sont élevés d'une valeur aléatoire et tous les points de l'autre côté sont rabaissés de cette même valeur. On choisit le nombre d'itérations que l'on souhaite effectuer pour obtenir un résultat plus ou moins précis.
De même que pour le MPD on peut appliquer un facteur de rugosité qui diminue à chaque itération pour contrôler la valeur aléatoire. Une fois terminé, on peut appliquer un filtre de lissage pour que les courbures du relief soient plus douces.



(1) (2) (3)

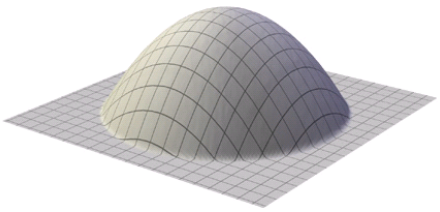
FIG. 1.24 – Itérations du Fault Algorithm

(1) :Fault Formation (itération $n°1$)
(2) :Fault Formation (itération $n°5$)
(3) :Fault Formation (après lissage)

1.3.7 Le Circles Algorithm

Le Circles Algorithm [LAU93] fonctionne lui aussi de manière très simple. Le principe est de disposer aléatoirement sur le terrain des cercles de rayon aléatoire. Chaque point se trouvant

dans un cercle est élevé de manière à former un dôme. Les points en dehors du cercle quant à eux gardent leur hauteur.



Altération de la hauteur des points dans le cercle

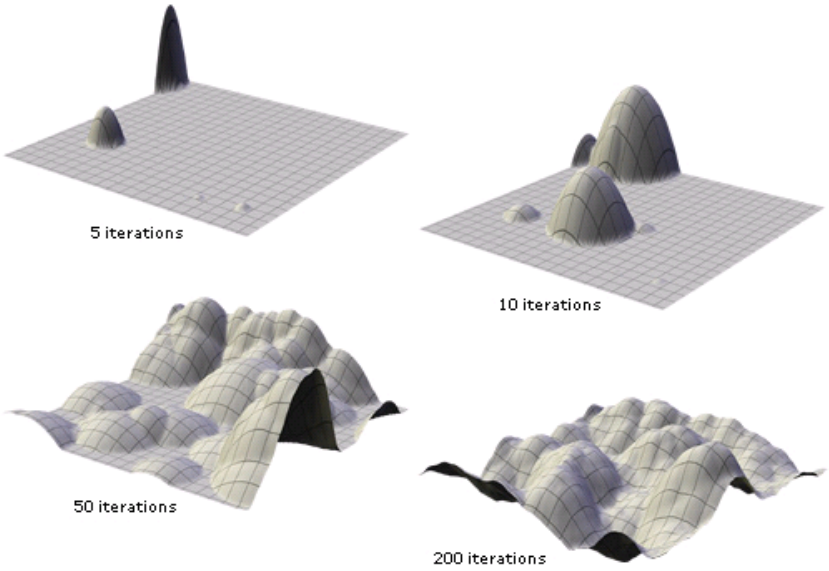


FIG. 1.25 – Itérations du Circles Algorithm

1.3.8 La triangulation de Delaunay

Il existe plusieurs algorithmes qui génèrent, de façon incrémentale, la triangulation de Delaunay [STE96]. Lorsqu'on part d'une triangulation et, en ajoutant un point, on veut recalculer les triangles. on va mettre en place deux algorithmes : la Destruction-Construction et le Flip Algorithme.

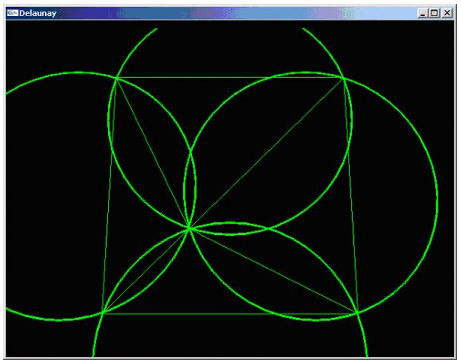


FIG. 1.26 – Les cercles circonscrits

Un des problèmes dans ces algorithmes est de trouver le triangle dans lequel se trouve le point qui sera ajouté. En choisissant d'insérer le point dans le triangle qui a la plus grande aire. La position du point est le barycentre des sommets du triangle (éventuellement pondérés). Les listes des triangles seront triées en fonction de leur aire (de façon décroissante pour que le plus grand triangle soit en début de liste).

Triangulation par Destruction/Construction Partant de quatre points de départ formant deux triangles, on utilise un triangle et un point choisi comme indiqué ci-dessus, puis tous les triangles dont le cercle circonscrit contient le point considéré sont détruits. Une fois les triangles illégaux détruits, on construit ceux qui sont engendrés par ce nouveau point, en ajoutant des arêtes reliant au point inséré les extrémités de celles qui n'appartenaient qu'à un seul triangle présent dans la liste de ceux qui ont été supprimés.

On utilise donc principalement deux méthodes : une fonction récursive qui détermine les triangles à éliminer, l'autre les triangles à construire. Voici l'algorithme général en pseudo-code :

Algorithme général :

$\rightarrow$ *Fonction inserePoint*()

$\rightarrow$ {

$\rightarrow$//On choisit le plus grands triangles

$\rightarrow$ *Triangle t* = *PremierTriangle*();

$\rightarrow$//Choisir un point dans le triangle

$\rightarrow$ *Point p* = *ChoisirPoint*(*t*);

$\rightarrow$//Initialiser une liste vide pour la destruction

$\rightarrow$ *Liste ld* = *InitialiseListeVide*();

→ $lc = InitialiseListeVide()$;

→//On detruit les triangles qui posent problemes

→ $detruireTriangles(p, t, ld)$;

→//On construit les triangles

→ $construireTrianges(p, ld, lc)$;

→//On ajoute les triangles

→ $AjouteTriangles(lc, lt)$;

→}

Destruction des triangles :

→ $Fonction\ detruireTriangles(Point\ p, Triangle\ t, Liste\ ld)$

→ {

→//Ajoute le triangle t a la liste l

→ $AjouteTriangle(l, t)$;

→//On regarde les triangles voisins

→ *Pour chaque triangle tv voisin de t*

→ *Si le triangle tv n'appartient pas a ld alors*

→ *Si p est inclu dans le cercle circonscrit de tv alors*

→ $detruireTriangles(p, tv, ld)$

→ $FinSi$

→ $FinSi$

→}

Construction des triangles :

→ $Fonction\ construireTriangles(Point\ p, Liste\ ld, Liste\ lc)$

→ {

→ *Pour chaque triangle t de la liste ld*

→ *Pour chaque voisin tv du triangle t*

→ *Soit p1, p2 les deux points du côté en commun avec t et tv*

→ $nouveaut = CreerTriangle(p, p1, p2)$

→ //On met a jour l'adjacence entre les triangles tv et $nouveaut$

→ $GererAdjacence(tv, nouveaut)$

→ //Ajouter le triangle $nouveaut$ dans une liste lc

$\rightarrow$ $AjouteTriangle(lc, nouveaut)$

$\rightarrow$ $FinPour$

$\rightarrow FinPour$

$\rightarrow$//Gerer les adjencences entre les triangles

$\rightarrow GererAdjencenceEntreTriangles(lc)$

$\rightarrow\}$

L'algorithme est assez simple. la plus grande difficulté est de ne pas se tromper avec les points et les triangles voisins.

Le Flip Algorithm Cet algorithme est plus rapide que le premier puisqu'il ya, en principe, moins de travail à faire. C'est une grande fonction récursive (qui pourrait être implémentée en itératif), qui vérifie si les côtés sont légaux. Un côté est défini comme étant l'arête commun à deux triangles. Pour savoir s'il est illégal, on regarde les points qui ne sont présents que dans un des triangles (donc les points opposés). L'image suivante montre un côté légal entre deux triangles, comme on le voit, les cercles circonscrits ne contiennent pas le point opposé.

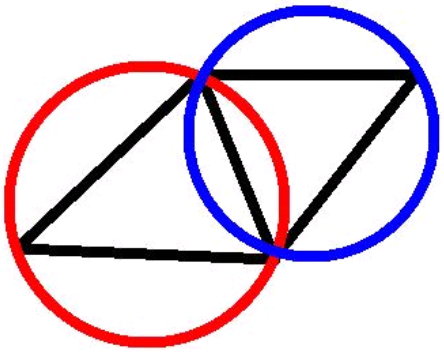


FIG. 1.27 – Un côté légal

Un côté est considéré comme étant illégal si au moins un de ces deux points est contenu dans le cercle circonscrit du triangle opposé.

Si c'est le cas, le côté illégal est remplacé par un côté relié par les deux points opposés. Et on testera les quatre côtés qui restent pour voir si ce changement a provoqué un autre conflit. L'image qui suit montre un côté illégal et le flip qui permet de rendre ce côté légal :

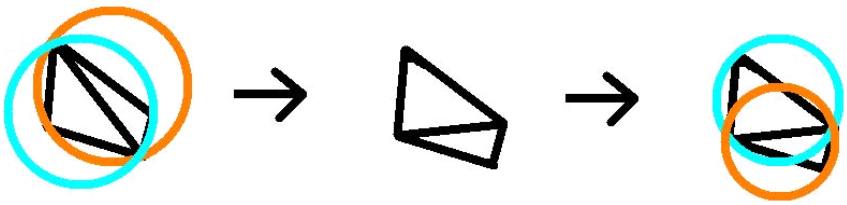


FIG. 1.28 – Un côté illégal (gauche), le flip (droite)

Donc l'algorithme peut être décrit par :

- Insérer le point, détruire le triangle le contenant
- Créer trois triangles qui ont ce nouveau point comme sommet
- Vérifier chaque côté externe pour une illégalité.

La vérification des côtés peut être donné par :

Algorithme de vérification

→ *Pour les deux points opposés, vérifier s'ils sont dans le cercle circonscrit du triangle opposé*
→ *Faire*
→ *Si le point est inscrit dans le cercle circonscrit du triangle opposé alors*
→ *Inverser le côté.*
→ *Vérifier les quatre côtés externes pour une illégalité.*
→ *FinSi*
→ *FinPour*

Comme on voit c'est donc un algorithme assez facile mais il faut faire attention à la structure de données qu'on utilise pour l'implémenter. Si on utilise une structure approprié pour cet algorithme alors le rendu risque d'être plus difficile. De l'autre côté, utiliser une structure idéale pour le rendu risque de rendre l'implémentation de ces algorithmes plus difficiles.

Néanmoins, on a choisi de favoriser le rendu du terrain puisqu'une fois le terrain créé, on n'a plus que le rendu à gérer. Donc l'implémentation de la triangulation sera plus difficile à comprendre et à implémenter.

1.3.9 La modélisaton des terrains par l'algorithme de Perlin

Pour générer un terrain aléatoire, la première idée (naïve) qui vient à l'esprit est de générer une hauteur aléatoire pour chaque position du terrain. Ceci donne des résultats très inexploitables. En effet, le terrain généré n'a aucune cohérence, il ressemble à tout sauf à un terrain.

En donnant une valeur entièrement aléatoire à chaque position du terrain, on obtient le résultat suivant :

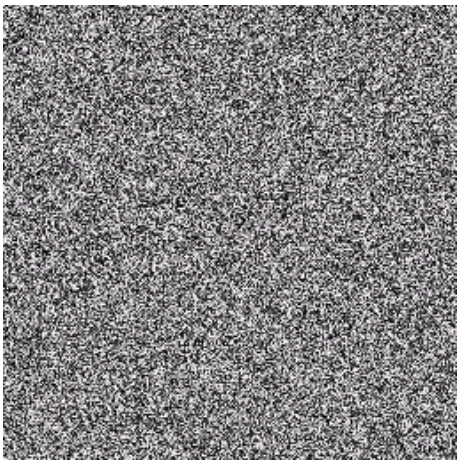


FIG. 1.29 – Terrain purement aléatoire

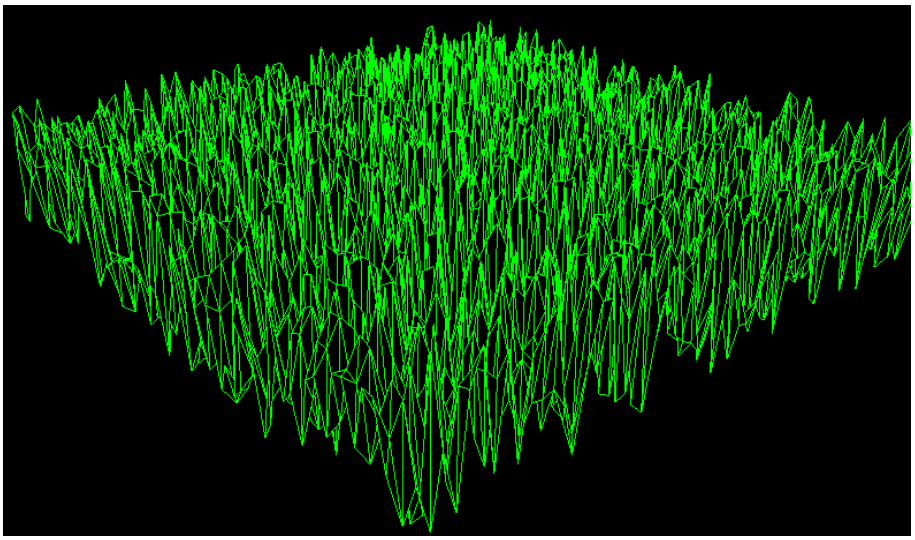


FIG. 1.30 – Même terrain dans le viewer de terrain de Fearyourself [STE96]

L'idée de l'algorithme de Perlin [KEN02] est d'affiner un terrain par itérations successives tout en interpolant entre les valeurs connues. La génération commence en fixant quelques valeurs puis en interpolant tous les autres points à partir de ces valeurs. A ce stade là, le résultat commence à ressembler à un terrain, mais il n'est pas naturel. Il va falloir ensuite lui ajouter des "calques" successifs pour affiner la pente du terrain et lui donner un aspect réaliste [KEN02].

La structure de données utilisée est celle d'un "calque". Un calque va représenter une image en niveau de gris, c'est sur un calque qu'on fera toutes les interpolations. Chaque calque sera aussi muni d'un attribut noté persistance qui définira un niveau sur les données de calque [KEN02].

```
→ struct calque{
→       int  **v;
→       int  taille;
→       float  persistance;
→};
```

Les données sont stockées dans une matrice d'entiers. Seront des entiers compris entre 0 et 255, image en niveau de gris oblige. On a néanmoins choisis le type int pour pouvoir gérer correctement les cas où les valeurs seraient amenées à dépasser 255, pour éviter les dépassements de capacité et les résultats aberrants.

Les terrains générés seront toujours carrés, c'est pourquoi on n'a besoin que d'un seul attribut taille.

Les fonctions de création et de destruction de calques sont données par le pseudo-code :

```
→ struct calque* init_calque(int t, float p){
→    struct calque *s = malloc(sizeof(struct calque));
→    s->v = malloc(t*sizeof(int*));
→    int i, j;
→    for (i = 0; i < t; i++){      /* réservation de l'espace pours les t*t valeurs */
→       s->v[i] = malloc(t*sizeof(int));
→       for(j = 0; j < t; j++)
→          s->v[i][j] = 0;       /* initialisation des valeurs par 0 */
→    }
→    s->taille = t;
→    s->persistance = p;
→    return s;}

→ void free_calque(struct calque *s){
→    int j;
→    for(j = 0; j < s->taille; j++)
→       free(s->v[j]);
→    free(s->v);
```

$\rightarrow\}$

Puis l'enregistrement de notre calque dans une image. On aura aussi besoin d'un générateur de nombres aléatoires (ou pseudo aléatoires).

/* génère un entier entre 0 et *a* */

$\rightarrow$ *unsigned char aleatoire*$(float\ a)\{$

$\rightarrow$ $\quad return(float)rand()/RAND\_MAX * a;$

$\rightarrow\}$

On va commencer par générer un calque entièrement aléatoire, sans aucune cohérence, ce calque nous servira à stocker toutes les valeurs aléatoires.

On va ensuite choisir certains points régulièrement espacés dans le calque pour prendre les valeurs aléatoires associées. Toutes les positions entre ces points choisis seront interpolées (linéairement ou autrement). On voit déjà apparaître l'un des paramètres de l'algorithme : le caractère régulièrement espacé des points choisis.

Ce paramètre est noté la fréquence. Choisir une fréquence de 2 signifie qu'on prend 9 points sur notre calque aléatoire pour commencer les interpolations. En effet, on coupe un segment en 2 parties, on a donc 3 points (le premier, celui du milieu et le dernier). Et comme on se place en deux dimensions, on a 3×3 valeurs à prendre. Une fois que tout le calque aura été interpolé entre ces $(fréquence + 1)^2$ valeurs, on va créer un autre calque avec d'autres interpolations, plus fines.

Chaque nouveau calque sera créé exactement de la même manière que le premier, à la différence près qu'on prend une fréquence située un octave au dessus de la fréquence associée au calque précédent. Il faut donc connaître la fréquence de chacun des calques et la fréquence de départ. La fréquence d'un calque étant la fréquence du calque précédent x la fréquence de départ.

Ainsi, avec une fréquence de départ de 2, le premier calque aura une fréquence de 2, le second aura 4, le troisième aura 8, puis 16, 32 ...et ainsi de suite. On voit ici apparaître un autre paramètre de notre algorithme : le nombre de calques à utiliser. Ce paramètre est en fait un nombre d'octaves à utiliser.

Notons qu'il n'est pas nécessaire que toutes les valeurs choisis aléatoirement sur un calque le soient aussi sur le suivant. L'essentiel étant que les calques correspondent à des interpolations de plus en plus fines, même s'ils ne correspondent pas aux même valeurs aléatoires choisies.

Et la partie astucieuse de l'algorithme consiste à additionner tous ces calques de manière pondérée, de sorte à ce que les calques interpolés finement influent moins que ceux interpolés grossièrement. C'est justement cette pondération qui permet de générer des montagnes, des vallées ...

On voit apparaître le dernier paramètre de notre algorithme : les pondérations des calques. C'est ce que on a appelé persistance dans la définition d'un calque.

Pour avoir un résultat correct, la persistance doit être comprise entre 0 et 1. Une persistance de 0.4 signifie que le premier calque sera pondéré de 40%, que le suivant sera pondéré de $0.4 \times 0.4 = 16\%$, le suivant de $0.4 \times 0.4 \times 0.4 = 6.4\%$ ...

Remarque : Une persistance supérieure à 0.5 amènera des incohérences si on prend trop d'octaves. En effet, pour une persistance de 0.9, le premier calque sera pondéré de 90%, le second de 81%, et leur somme dépasse 100%, d'où une valeur supérieure à 255 pour un pixel, d'où aberration. Pour pallier ce problème, on a rajouté un traitement final pour ramener les valeurs supérieures à 255 dans un intervalle correct. Mais une persistance supérieure à 0.5 n'amènera pas toujours des incohérences, tout dépend du nombre d'octaves. Bref, il y aura incohérence s'il y a trop d'octaves par rapport à la persistance (quand elle est supérieure à 0.5).

Création du calque aléatoire Pour créer un calque on a besoin d'un générateur de nombres aléatoires et d'utiliser le pseudo-code suivant :

$\rightarrow$ *struct calque* $*$ *random;*
$\rightarrow$ $random = init\_calque(taille, 1);$
$\rightarrow$ $for(i = 0; i < taille; i++)$
$\rightarrow$ $\quad for(j = 0; j < taille; j++)$
$\rightarrow$ $\quad\quad random->v[i][j] = aleatoire(256);$ $\quad /* valeurs\ du\ calque\ aléatoire */$

Le calque aléatoire n'a pas besoin de persistance, la valeur passée au constructeur (1) ne sera jamais utilisée.

Remplissage des calques Chaque calque agira sur le calque aléatoire avec sa propre fréquence, qui changera d'un octave à l'autre.

$\rightarrow$ $for(n = 0; n < octaves; n++)\{$

```
→       for(i = 0; i < taille; i++)
→           for(j = 0; j < taille; j++){
→               a = valeur_interpolee(i, j, f_courante, random);
→               mes_calques[n]-> v[i][j] = a;    /* valeurs des calques du travail */
→           }
→       f_courante* = frequence;
→}
```

Le traitement effectué par ces boucles imbriquées consiste à aller récupérer les valeurs de chacune des positions du calque en cours de traitement. Ces valeurs viennent d'une interpolation. L'interpolation dépend du calque aléatoire (c'est lui qui contient les valeurs de base), de la fréquence $f\_courante$ (pour déterminer quelles valeurs du calque aléatoire on va prendre) et de la position (i, j) en cours.

Interpolation des valeurs On doit choisir certaines valeurs sur le calque aléatoire, puis interpoler entre ces valeurs. La fréquence va découper notre calque en plusieurs parties. Tout d'abord, il faut déterminer dans quelle partie le point (i, j) est situé, puis récupérer les valeurs du calque aléatoire aux positions qui délimitent cette partie, puis interpoler entre ces valeurs.

La détermination des bornes entourant le point recherché se fait par division entière en fonction de la taille d'un intervalle.

```
→ int valeur_interpolee(int i, int j, int frequence, struct calque *r){
→       /* déterminations des bornes */
→       int borne1x, borne1y, borne2x, borne2y, q;
→       float pas;
→       pas = (float)r-> taille/frequence;

→       q = (float)i/pas;
→       borne1x = q * pas;
→       borne2x = (q + 1) * pas;

→       if(borne2x >= r-> taille)
→          borne2x = r-> taille - 1;
```

$\rightarrow$ $q = (float)j/pas;$

$\rightarrow$ $borne1y = q * pas;$

$\rightarrow$ $borne2y = (q+1) * pas;$

$\rightarrow$ $if(borne2y >= r->taille)$

$\rightarrow$ $\quad borne2y = r->taille - 1;$

$\rightarrow$ $/* récupérations\ des\ valeurs\ aléatoires\ aux\ bornes */$

$\rightarrow$ $int\ b00, b01, b10, b11;$

$\rightarrow$ $b00 = r->v[borne1x][borne1y];$

$\rightarrow$ $b01 = r->v[borne1x][borne2y];$

$\rightarrow$ $b10 = r->v[borne2x][borne1y];$

$\rightarrow$ $b11 = r->v[borne2x][borne2y];$

$\rightarrow$ $int\ v1 = interpolate(b00, b01, borne2y - borne1y, j - borne1y);$

$\rightarrow$ $int\ v2 = interpolate(b10, b11, borne2y - borne1y, j - borne1y);$

$\rightarrow$ $int\ fin = interpolate(v1, v2, borne2x - borne1x, i - borne1x);$

$\rightarrow$ $return\ fin;$

$\rightarrow$ $\}$

Une fois les valeurs du calque aléatoire récupérées aux bornes de notre intervalle, on doit faire une interpolation bilinéaire. C'est pourquoi on a 3 appels à la fonction d'interpolation.

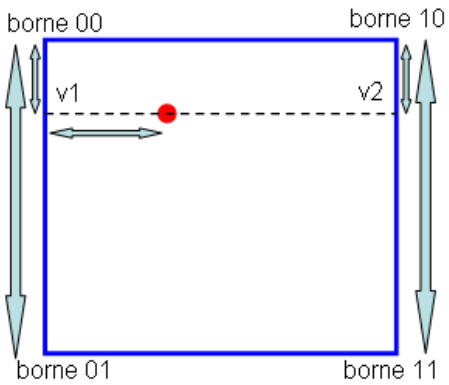


FIG. 1.31 – Interpolation bilinéaire

On connait les valeurs des bornes (elles viennent directement du calque aléatoire), on interpole entre ces valeurs pour trouver les valeurs $v1$ et $v2$, puis on interpole entre $v1$ et $v2$ pour avoir la valeur au point recherché.

On peux interpoler de la manière qu'on veux, linéairement ou pas. Voici le code pour une interpolation linéaire et pour une non linéaire :

$\rightarrow int\ interpolate(int\ y1, int\ y2, int\ n, int\ delta)\{$
$\rightarrow \quad if(n! = 0)$
$\rightarrow \quad\quad return\ y1 + delta * (y2 - y1)/n;$
$\rightarrow \quad else$
$\rightarrow \quad\quad return\ y1;$
$\rightarrow\}$

Les valeurs d'entrée étant les valeurs connues entre les quelles on va interpoler $(y1\ et\ y2)$, l'intervalle entre les valeurs connues (n) et l'intervalle entre la première valeur connue et la valeur recherchée $(delta)$.

Cette interpolation donne des résultats très rapides, très (trop) réguliers, les pentes sont parfaites.

Et voici un code d'une interpolation non linéaire :

$\rightarrow int\ interpolate(int\ y1, int\ y2, int\ n, int\ delta)\{$
$\rightarrow \quad if(n == 0)$
$\rightarrow \quad\quad return\ y1;$
$\rightarrow \quad if(n == 1)$
$\rightarrow \quad\quad return\ y2;$

$\rightarrow$ $float\ a = (float)delta/n;$

$\rightarrow$ $float\ v1 = 3 * pow(1-a,2) - 2 * pow(1-a,3);$

$\rightarrow$ $float\ v2 = 3 * pow(a,2) - 2 * pow(a,3);$

$\rightarrow$ $return\ y1 * v1 + y2 * v2;$

$\rightarrow\}$

Les valeurs d'entrée sont les mêmes. Les figures suivante représentent un exemple interpolé avec ces deux méthodes. Les points de base sont marqués en rouge.

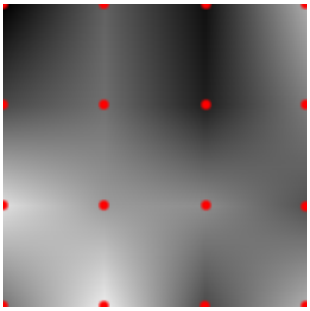


FIG. 1.32 – Interpolation linéaire

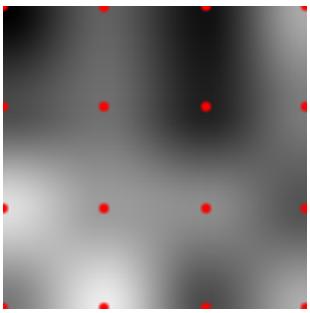


FIG. 1.33 – Interpolation non linéaire

On remarque tout de suite que cette interpolation non linéaire nous donne de biens meilleurs résultats, même si elle est un peu plus longue à calculer. Le terrain est beaucoup plus arrondi.

Ajout de tous les calques Une fois que tous les calques ont été déterminés, il nous reste à les ajouter en tenant compte de la persistance de chacun. C'est à ce moment là que peuvent surgir les incohérences. Si l'addition finale dépasse 255, le résultat sera aberrant. Voici un exemple pour bien vous rendre compte du risque :

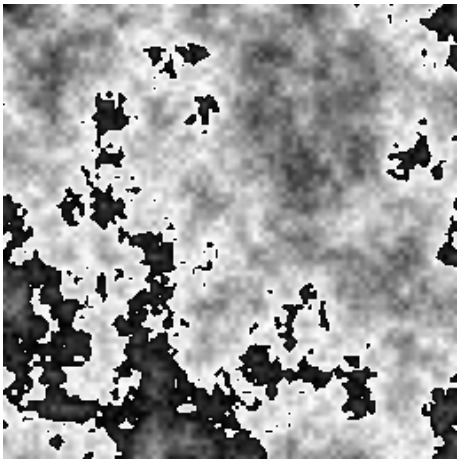


FIG. 1.34 – Résultat incohérent

Les valeurs supérieures à 255 repassent à 0 lorsqu'elles sont enregistrées. C'est la raison pour la quelle il faut surveiller les valeurs obtenues et les corriger si besoin est.

$\rightarrow$ /* *calcul de la somme de toutes les persistances, pour ramener les valeurs dans un intervalle acceptable* */

$\rightarrow$ *sum_persistances* = 0;

$\rightarrow$ $for(i = 0; i < octaves; i++)$

$\rightarrow$ *sum_persistances*+ = *mes_calques*[*i*]− > *persistance*;

$\rightarrow$ /* *ajout des calques successifs* */

$\rightarrow$ $for(i = 0; i < taille; i++)$

$\rightarrow$ $for(j = 0; j < taille; j++)\{$

$\rightarrow$ $for(n = 0; n < octaves; n++)$

$\rightarrow$ $c- > v[i][j]+ = mes\_calques[n]- > v[i][j] * mes\_calques[n]- >$ *persistance*;

$\rightarrow$ /* *normalisation* */

$\rightarrow$ $c- > v[i][j] = c- > v[i][j]/sum\_persistances;$

$\rightarrow$ }

On obtient donc ce résultat réaliste

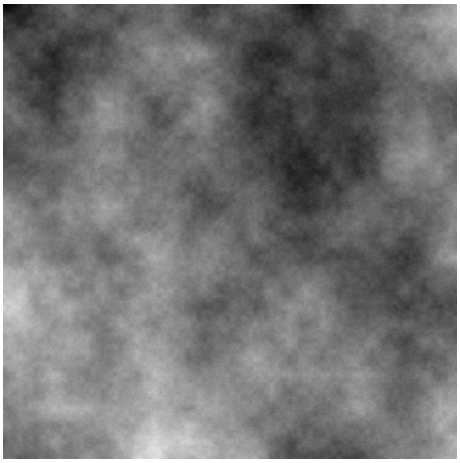


FIG. 1.35 – Résultat cohérent

Et voici, le code pour lisser ce terrain :

```
→ /*lissage*/
→ struct calque *lissage;
→ lissage = init_calque(taille, 0);
→ for(x = 0; x < taille; x++)
→    for(y = 0; y < taille; y++){
→       a = 0;
→       n = 0;
→       for(k = x − 5; k <= x + 5; k++)
→          for(l = y − 5; l <= y + 5; l++)
→          if((k >= 0)&&(k < taille)&&(l >= 0)&&(l < taille)){
→             n++;
→             a+ = c− > v[k][l];
→          }
→       lissage− > v[x][y] = (float)a/n;
→    }
```

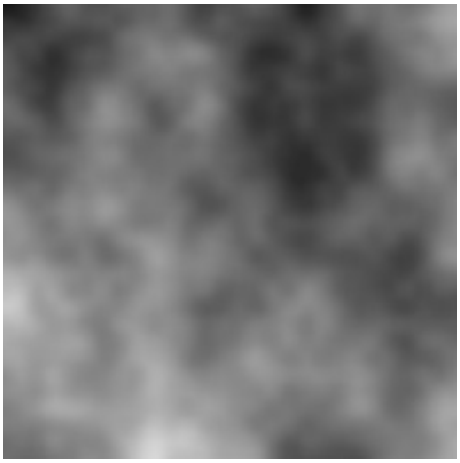


FIG. 1.36 – Résultat lissé

1.4 Bilan

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

Méthode	Caractéristiques	Avantages	inconvénients
Subdivision en triangles	– Basé sur une subdivision récursive du modèle. – L'introduction d'un facteur aléatoire.	– Méthode simple à implémenter – Possibilité de contrôle du résultat via le paramètre de rudesse.	– Apparence des rainures entre les arêtes des triangles. – Coûteuse en terme de calculs.
Diamond-Square	– Processus itératif. – La hauteur d'un point dépend des points voisins.	– Possibilité de contrôler le résultat via le coefficient de lissage. – Problème de rainures réglé	– L'apparence de petites crêtes (sommets) surtout pour les terrains lisses.
Transformée de fourier	– Technique spectrale. – Basée sur la FFT 2-dimensionnelle.	– Algorithme simple n'est pas un processus itératif. – Pas de rainures artificielles ni des crêtes.	– Un coté est similaire au coté en face.
Technique de multiplication	– Caractéristiques de la technique choisie (Diamond-Squart, fourier,...).	– Simulation de processus de l'érosion (vallées plus lisses que crêtes).	– Deux fois plus lente que la technique choisie.

Mid-point displacement	– Basé sur une subdivision récursive.	– On est certain du paysage que l'on va représenter. – Possibilité de retoucher et d'ajouter des décors.	– Prend de l'espace pour stocker les informations.
Fault Formation	– Faites d'une manière itérative. – Application du facteur de rugosité.	– Fonctionne d'une manière très simple.	– Lente pour avoir le résultat souhaité.
Circles Algorithm	– Faites d'une manière itérative. – Disposition des cercles de rayon aléatoire sur le terrain.	– Fonctionnement très simple.	– Prend un grand nombre d'itérations pour arriver au résultat souhaité.
Triangulation de Delaunay	– L'utilisation de la subdivision récursive.	– Possibilité de contrôler 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 coûteuse en terme de calculs.

1.5 Conclusion

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 ; l'algorithme de Perlin est lourd en terme de calcul et l'algorithme de Diamond-Square produit l'effet de L'apparence de petites crêtes.

Nous allons essayer par la suite d'éliminer l'effet de l'apparence de petites crêtes dans l'algorithme de Diamond-Square par l'utilisation d'un seul calque tel qu'il est utilisé dans l'algorithme de Perlin, et de voir si l'algorithme résultat : est plus efficace que Diamond-Square, et s'il est plus rapide et optimal que l'algorithme de perlin.

Chapitre 2

Génération de textures de terrain

2.1 Introduction

Lorsqu'on affiche un terrain en 3 dimensions, il faut qu'on arrive à le colorer ou à utiliser une texture pour le rendu des triangles générés dans la phase précédente (modélisation). Pour un rendu plus réel, la texture doit être appliquée correctement. Le problème réside dans la génération de cette texture, c'est donc ce que nous allons voir dans ce chapitre.
Cette partie montrera les idées générales sur la génération de textures, et servira a présenter la façon avec la quelle un objet (terrain) sera afficher en espérant obtenir un résultat suffisamment réaliste.

L'habillage consiste à prendre une image (texture) que l'on va " coller " sur les polygones déjà calculer.

2.2 Utilisation de larges textures

Une méthode possible serait l'utilisation d'une large texture [AND03] unique qui couvrirait tout le paysage. Cette texture pourrait être une photo aérienne du paysage qu'il suffirait de "plaquer" au sol.

Bien qu'en théorie l'utilisation de larges textures puisse fournir d'excellents résultats, car cette méthode est très facile à mettre en œuvre, l'idée a été très vite oubliée, car elle ne permet d'avoir suffisament de détails qui seront visibles lors du rendu.

2.3 Le mélange de textures de base

Lorsqu'on veut afficher un terrain en 3 dimensions, on utilise généralement une image qui permet d'avoir la hauteur de chaque point du terrain. Cette image, généralement monochrome, ressemble à ceci :

FIG. 2.1 – Image de niveaux

Bien sûr, la gestion des images de niveaux est une convention entre le programme et l'image de niveaux. Généralement, une couleur sombre représente une faible altitude dans le terrain et une couleur claire représente un niveau élevé.

Voici ce que donne l'affichage du terrain en 3 dimensions et en mode filaire :

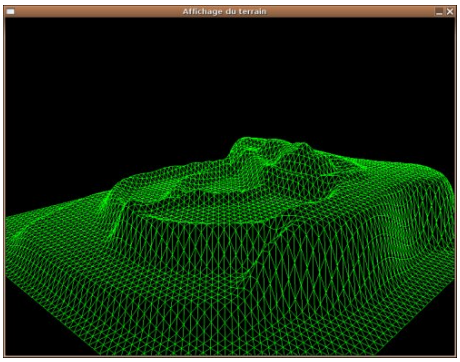


FIG. 2.2 – Affichage de terrain 3D en filaire

Mais comment décider de la façon avec la quelle seront colorier les différentes zones du terrain ? Une première solution serait d'utiliser des couleurs dépendant des hauteurs des sommets du terrain. C'est bien sûr possible , mais généralement cette méthode ne donne pas de très bons résultats.

Donc l'idée est de générer une texture qui sera faite parfaitement par rapport au terrain qui

sera affiché. Ce que qu'on veut faire consiste à utiliser trois textures différentes pour afficher correctement le terrain. Voici les trois images qu'on va utiliser :

FIG. 2.3 – Trois images de base

Donc on va calculer un mélange de ces trois images pour arriver à générer une texture qui sera à l'image de niveaux. On obtient donc ceci :

FIG. 2.4 – Texture générée à partir des trois images de base

Et lorsqu'on applique cette texture au rendu 3D, cela donne :

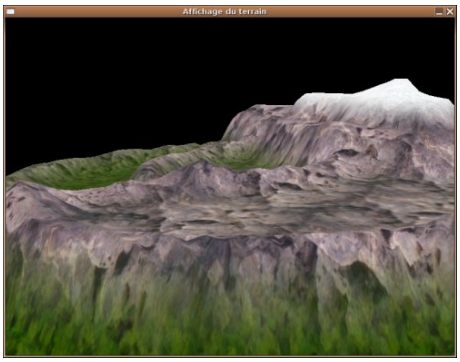


FIG. 2.5 – Terrain texturé

La prochaine section présentera comment on crée cette texture.

Un inconvénient à la méthode de mélange de textures de base [CLI95, AND03] est le fait qu'on

doit effectuer une passe par texture que l'on module. Il faut ainsi retracer le paysage autant de fois qu'il y a de textures de base différentes. Cela signifie que lors de la première passe on trace le paysage avec la texture qui représente l'herbe, et ensuite on retrace le paysage avec la texture qui représente les rochers. Heureusement, en pratique, effectuer cette opération n'est pas deux fois plus lente comme on pourrait le penser.

Pour générer un terrain comme celui-ci, l'algorithme qu'on va utiliser est relativement simple :

$\rightarrow$ *Pour chaque pixel de la texture du terrain*
$\rightarrow$ *Faire*
$\rightarrow$ *Soit h la hauteur dans le terrain du pixel associé*
$\rightarrow$ *Utiliser la valeur de h pour calculer la partie des trois textures de base*
$\rightarrow$ *Affecter la couleur au pixel*
$\rightarrow$ *Fin*

Il faut donc savoir calculer la hauteur h du pixel associé et calculer la couleur du pixel.

2.3.1 Calcul de la hauteur h

Pour calculer la hauteur du pixel associé, on utilise l'image de niveaux. Pour le cas le plus simple, la taille de la texture du terrain est la même que l'image de niveaux. C'est donc facile d'avoir la hauteur du pixel associé.

Comme le montre ce code, on peut facilement passer de l'échelle $0 - 255$ de l'image de niveaux à l'intervalle $[a, b]$.

Avant de montrer l'algorithme général pour cette transformation, remarquons qu'en utilisant uniquement une couleur d'une image bitmap de niveaux, on a seulement 256 niveaux.

En utilisant, les trois couleurs à la fois, on peut gérer plus d'un million de niveaux différents. Mais dans le cas général, un seul octet suffit pour faire un joli terrain.

Calcul de la hauteur :

$\rightarrow$ */ * C'est monochrome, on peut prendre n'importe quelle couleur et c'est une valeur entre* $0 - 255$ ** /*
$\rightarrow h =$ *la valeur du pixel* (i, j)*;*

$\rightarrow h\ /=\ 255.0;\ /*\ Valeur\ entre\ [0,1]\ */$

$\rightarrow h*=b-a;\ \ /*\ Valeur\ entre\ [0,b-a]\ */$

$\rightarrow h+=a;\ \ \ /*\ Valeur\ entre\ [a,b]\ */$

On a donc la solution pour récupérer la hauteur du pixel associé et la transformer à n'importe quelle échelle.

2.3.2 Calcul de la couleur du pixel

Pour avoir la couleur du pixel, on utilise directement la hauteur du pixel. Dépendant de la hauteur, on va utiliser un mélange entre les différentes images de niveaux.
L'image suivante montre le dégradé des images de base :

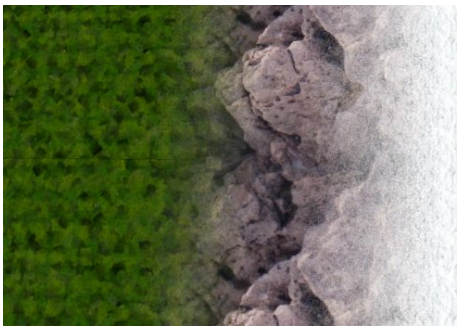


FIG. 2.6 – Dégradé des images de base

En effet, si la hauteur est assez basse, on utilisera uniquement la couleur de l'image qui représente de l'herbe. Ensuite, si la hauteur est très haute alors on utilise la couleur de l'image de neige. Bien sûr, cela veut dire qu'on peut faire des mélanges entre les trois images. Voici la fonction qu'on utilise pour donner la couleur du pixel :

La fonction va donc remplir un tableau à 3 éléments qui donnera la participation de la couleur de chacune des trois images de base. Si la valeur est à *zéro*, on n'utilise pas cette image. Si la valeur est à 1.0, on utilise entièrement cette composante.

Calcul de la composition des couleurs :

$\rightarrow\ /*\ Cette\ fonction\ remplit\ le\ pourcentage\ que\ ce\ pixel\ doit\ avoir\ dependant\ de\ l'hauteur\ associe$

$\rightarrow\ *\ Si\ l'hauteur\ est\ faible,\ c'est\ que\ de\ l'herbe$

$\rightarrow\ *\ Sinon\ c'est\ un\ mélange\ herbe\ -\ roche$

```
→ * Plus haut, c'est que de la roche
→ * Ensuite un mélange roche − neige
→ * Et enfin, c'est que de la neige
→ */

→ void Terrain_RemplitPerc(float *perc, unsigned char haut)
→ {
→       /* Que de la prairie */
→       if(haut < 60)
→             {
→             perc[0] = 1.0f;
→             perc[1] = 0.0f;
→             perc[2] = 0.0f;
→             }
→       /* Mélange entre prairie et roche */
→       else if(haut < 130)
→             {
→             perc[0] = 1.0f - (haut - 60.0f)/70.0f;
→             perc[1] = (haut - 60.0f)/70.0f;
→             perc[2] = 0.0f;
→             }
→       /* Que de la roche */
→       else if(haut < 180)
→             {
→             perc[0] = 0.0f;
→             perc[1] = 1.0f;
→             perc[2] = 0.0f;
→             }
→       /* Mélange entre roche et la neige */
→       else if(haut < 220)
→             {
→             perc[0] = 0.0f;
```

$\rightarrow$ $perc[1] = 1.0f - (haut - 180.0f)/40.0f;$

$\rightarrow$ $perc[2] = (haut - 180.0f)/40.0f;$

$\rightarrow$ }

$\rightarrow$ $/* \ Que\ de\ la\ neige \ */$

$\rightarrow$ $else$

$\rightarrow$ {

$\rightarrow$ $perc[0] = 0.0f;$

$\rightarrow$ $perc[1] = 0.0f;$

$\rightarrow$ $perc[2] = 1.0f;$

$\rightarrow$ }

$\rightarrow$}

Les valeurs utilisées ont été trouvées empiriquement. On joue avec les valeurs pour trouver de meilleurs résultats.

La somme des trois valeurs doit être égale à 1. Si on dépasse cette limite, on risque d'avoir des couleurs saturées.

Supposons pour l'instant qu'on est dans le cas où la texture générée possède la même taille que les trois images de base. Il est donc facile de savoir le pixel associé aux images de base.

Une fois qu'on a la participation des couleurs des trois images, on peut calculer la couleur du pixel. L'algorithme général est :

Calcul de la couleur :

$\rightarrow$ *Calcul du pixel* (i, j) :

$\rightarrow$ *Appel de Terrain_RemplitPerc*$(perc, hauteur\ du\ pixel(i, j))$.

$\rightarrow$ *Récupération du pixel* (i, j), *nommé* A, *de l'image prairie associé.*

$\rightarrow$ *Récupération du pixel* (i, j), *nommé* B, *de l'image roche associé.*

$\rightarrow$ *Récupération du pixel* (i, j), *nommé* C, *de l'image neige associé.*

$\rightarrow$ *Rendre* $perc[0] * A + perc[1] * B + perc[2] * C$.

Ce qui se traduit par :

Calcul de la couleur (en C) :

$\rightarrow$ /* *Recuperation des participations des couleurs pour le pixel courant* */

$\rightarrow$ $Terrain\_RemplitPerc(perc, ((unsigned\ char*)image- > pixels)[tmpi * image- > w * 3 + tmpj * 3]);$

$\rightarrow$ /* *Recuperation des couleurs par image de base* */

$\rightarrow$ $b = perc[0] * Terrain\_GetPixelColor(prairies, tmpi, tmpj, 0);$

$\rightarrow$ $g = perc[0] * Terrain\_GetPixelColor(prairies, tmpi, tmpj, 1);$

$\rightarrow$ $r = perc[0] * Terrain\_GetPixelColor(prairies, tmpi, tmpj, 2);$

$\rightarrow$ $b+ = perc[1] * Terrain\_GetPixelColor(rocheuses, tmpi, tmpj, 0);$

$\rightarrow$ $g+ = perc[1] * Terrain\_GetPixelColor(rocheuses, tmpi, tmpj, 1);$

$\rightarrow$ $r+ = perc[1] * Terrain\_GetPixelColor(rocheuses, tmpi, tmpj, 2);$

$\rightarrow$ $b+ = perc[2] * Terrain\_GetPixelColor(neige, tmpi, tmpj, 0);$

$\rightarrow$ $g+ = perc[2] * Terrain\_GetPixelColor(neige, tmpi, tmpj, 1);$

$\rightarrow$ $r+ = perc[2] * Terrain\_GetPixelColor(neige, tmpi, tmpj, 2);$

On a bien une composition des trois images de base. La fonction *Terrain_GetPixelColor* prend en paramètre un couple $(tmpi, tmpj)$ pour rendre une couleur (0 pour bleu, 1 pour vert et 2 pour rouge) de l'image passé en premier paramètre. Le dernier paramètre permet de spécifier si on veut le canal rouge, vert ou bleu.

2.3.3 Taille de la texture

Il est évident qu'il n'est pas obligatoire que la texture soit de la même taille que les images de base. Pour résoudre ce problème, on utilise le modulo qui permettera de ne plus avoir de problèmes de débordement. Cette solution n'est pas parfaite mais elle permet d'avoir un meilleur résultat.

Pourquoi avoir une taille plus grande ? Cela permet d'avoir plus de détail dans le terrain. C'est encore un compromis entre le temps de calcul, la consommation mémoire et les détails qui seront visibles lors du rendu.

Voici une comparaison entre une texture de taille $1024 * 1024$(à droite) et une texture de taille $4096 * 4096$(à gauche). On voit parfaitement bien la différence de détail.

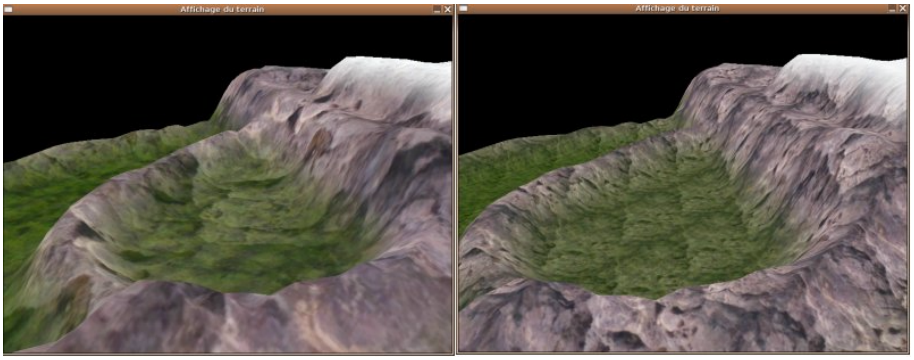


FIG. 2.7 – Influence de la taille de la texture

2.4 Le multi-texturing

La méthode qu'on vient de décrire, s'appelle le multi-texturing [AND03] qui est supportée sur certaines cartes graphiques. En général, les cartes graphiques du marché supportant cette extension permettent d'appliquer deux textures en parallèle.

Ceci permet de réduire le nombre de passes nécessaires à l'affichage et améliore les performances du rendu graphique.

Cette méthode reste toutefois limitée car seules les valeurs de coordonnées de textures et les textures elles-mêmes peuvent être différentes. Plus précisément cela signifie qu'un seul niveau de transparence commun sera utilisé par les différentes textures, par exemple. On ne peut pas utiliser deux politiques différentes de transparence pour ces deux textures.

Les ombres sont calculées à l'aide d'une méthode simple, mais offrant des résultats très satisfaisant. En effet, on calcule pour chaque hauteur du terrain une intensité lumineuse en fonction de la pente par rapport au soleil. Ces valeurs sont alors stockées dans une texture que l'on dénomme "shadow-map".

2.5 Geo-mip-mapping et texture splatting

Nous allon dans ce qui suit décrire le geo-mip-mapping [MGD00]. Il s'agit d'une technique utilisée pour créer efficacement différents niveaux de détails pour les terrains. Le terrain est divisé en un ensemble de carreaux (on nomme cela le tiling) de puissance de deux. Chaque carreau possède 5 niveaux de détails différents avec un nombre de triangle allant de 512 à 2. Du fait de la structure efficace des niveaux des LoDs (Niveaux de détails). Il peuvent tous utiliser le même vertex buffer. La figure suivante illustre à quoi ressemblent les différents niveaux de détails.

FIG. 2.8 – Geo-mip-mapping

En fonction de la différence de hauteur du carreau et de la distance de la caméra, Ultimate 3D [MGD00] décidera quel niveau de détails utiliser pour afficher le carreau en question. Cette grande variété de LoDs rend le moteur de rendu de terrain incroyablement rapide. Un problème auquel on est fréquemment confronté lorsqu'on affiche un terrain, concerne la façon avec la quelle on obtient de bonnes textures pour le terrain. Chaque partie du terrain devrait avoir une couleur différente, mais une texture qui couvre l'ensemble du terrain serait soit extrêmement grande, soit très peu détaillée. Aucune des deux solutions n'est intéressante. Une technique pour éviter cela consiste en l'utilisation d'une carte de détails (detail map) [JOH07].

C'est pourquoi il existe une technique nommée texture splatting [MGD00]. Lorsque on utilise cette méthode, on peux utiliser autant de textures, étirées sur le terrain (cet étirement correspond aux tiling mais pour les textures). Il est donc, ensuite possible d'utiliser les maps alpha pour définir des transparences différentes des textures en différents points. L'image suivante en illustre le fonctionnement :

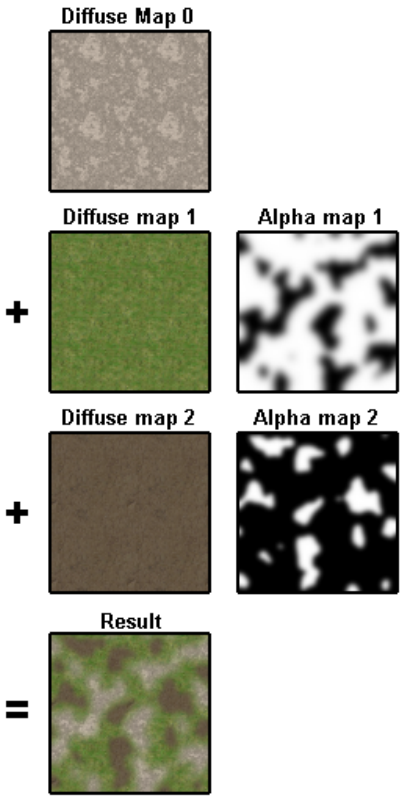


FIG. 2.9 – splatting

Le texture splatting nous donne la possibilité de rendre notre terrain vraiment détaillé et vivant. On peux utiliser une texture de pierre pour les collines et des textures d'herbe ou de boue pour les plateaux. Ainsi, on peut également mettre la neige sur les monts des montagnes et en plus, on aura la possibilité d'utiliser une texture séparée pour les chemins sur les quels les personnages peuvent marcher. On remarque que le texture splatting requière le support du multi texturing avec, au moins, deux textures simultanées.

2.6 Mélange de texture de terrain par le materiel

Il est possible d'implémenter les techniques ci-dessus dans le matériel [YX09] en utilisant le pipeline fixe de fonction par DirectX sans besoin d'un GPU programmable. Cependant, le pipeline fixe de fonction ne tient pas compte de la lecture explicite de différents canaux de couleur de texture dans le matériel, donc les maps alpha combinées ne seraient pas possibles en utilisant l le pipeline fixe de fonction et chaque texture de terrain besoin de posséder la map alpha dans la mémoire de texture.

Les GPUs programmables employant un shader Pixel tient compte de la lecture de différents

canaux de couleur de texture, c'est à dire qu'il emploie une map alpha combinée et la mémoire de sauvegarde de texture.

Pour réaliser le résultat désiré employant la canalisation fixée de fonction l'opération de texture $D3DBLEND\_INVSRCALPHA$ doit être exécuté.Cette opération fait ce qui suit :

$$ResultantColor = Alpha * Texture + (1 - Alpha) * PreviousColor$$

Cette opération est ce qui doit être mis en application sur le GPU programmable.

Pour HLSL (High Level Shading Language) [MA98] l'approche générale est comme suit :

- Produire Proceduralement un ensemble de maps alpha pour les textures désirées de terrain (ou charger des maps alpha pré générées à partir du disque).
- Charger l'ensemble de textures de terrain qui seront appliquées dans la mémoire.
- Initialiser le shader de Pixel.
- Placer les étapes de texture de terrain pour que le shader de Pixel mette en référence les maps alpha et les textures de terrain.
- Rendre la géométrie en utilisant le shader de Pixel.

2.7 Les détails textures

Nous présentons ici une dernière méthode, elle possède l'avantage d'être légèrement plus rapide et peut être avantageuse pour des configurations matérielles plus limitées.

Il s'agit de l'utilisation de "détails textures" [AND03]. Derrière ce nom se cache en fait une combinaison de ce que nous avons présenté. Il s'agit effectivement d'une méthode employant les larges textures, le mélange de texture et de multi-texturing (à condition que celui-ci soit supporté par la carte graphique 3D).

Comme nous l'avons vue, la méthode des larges textures possède comme avantage d'être simple à implémenter (pour peu qu'on dispose de la texture qu'on souhait appliquer au terrain), et ne nécessite qu'une seule passe pour tracer le paysage. L'inconvénient majeur est la consommation excessive de ressources mémoire. L'idée est de diminuer considérablement la taille de la texture d'habillage en atténuant la précision d'échantillonnage. On compensera cette perte de précision par l'adjonction d'une texture de détail lorsque l'observateur se rapproche du modèle texturé, évitant ainsi un flou ("blur") disgracieux.

Celle-ci sera unique pour tout le paysage et sera de petite taille (par exemple $512x512$).

Elle sera appliquée au paysage soit par simple multi-texturing, soit par une seconde passe. On effectuera cette opération de façon similaire à la méthode décrite pour le mélange de texture, excepté que nous ne ferons pas intervenir de valeur de transparence. L'apparence générale sera ainsi donnée par la texture d'habillage recouvrant l'entièreté du terrain.

Pour obtenir des résultats satisfaisants, la texture de détail sera encodée en tons de gris et représenteras des motifs à haute fréquence (craquelures, aspect rugueux, sillons, grains, . . .etc).

Cette texture de détail sera répétée un certain nombre de fois sur tout le paysage.

2.8 Conclusion

Nous avons vu dans ce chapitre un ensemble de techniques d'habillage de terrains 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. Dans cette technique la texture générée dépend des hauteurs, si on joue sur le critère profondeur, nous pourrons gérer le niveau de détails (LOD), alors la texture de terrain change avec le changement de point de vue, on revient alors a la technique de texture splatting, La combinaison entre les deux techniques est donc possibles.

Chapitre 3

Illumination temps réel de terrain

3.1 Introduction

Plusieurs jeux ont des scènes en plein air. Dans cet environnement l'objet dominant le plus visible est le terrain. Il y a plusieurs travaux publiés sur la modélisation de la géométrie des terrains mais pas sur leur illumination [NK01]. Dans cette section nous allons expliquer les différents facteurs qui contribuent dans l'illumination en plein air des terrains dans le monde réel, et présenter deux techniques pour la simulation de cette illumination sous le changement des condition de l'illumination [NK01].

3.2 Présentation de l' l'illumination en plein air

On distingue entre l'illumination directe qui vient directement sur le terrain depuis la source lumineuse et l'illumination indirecte qui est générée à partir des interactions de la lumière (des parties illuminées du terrain influent sur l'éclairement des autres). On se concentre sur l'illumination du jour, cependant des approches similaires sont utilisées pour la nuit [NK01].

La source lumineuse la plus importante de l'illumination directe est le soleil. Le soleil est très distant, donc on peut le considérer comme une source lumineuse directionnelle, et pas une source ponctuelle Il possède un petit mais non nul diamètre angulaire (près de demi degré). Ceci a conduit que les ombres produits par le soleil ont des frontières légères. L'intensité et la couleur de la lumière reçu à partir du soleil varient en fonction de l'heure et la saison. Aussi, les nuages réduisent la visibilité de soleil.

Le ciel est une autre source de l'illumination directe. Bien sur cette lumière à l'origine est

provoquée par le soleil et se diffuse des molécules de l'air, des goûtes de la pluie, d'autres impuretés de l'air, mais il faut le traité comme une deuxième source lumineuse indépendante du soleil. Le ciel est une source lumineuse a surface très large qui possède la couverture d'un hemisphere. L'intensité et la couleur de la lumière émise du ciel varie sur sa surface à n'importe quel instant donné, l'effet visible du lumière du ciel est plus clair sur les ombres quand il ne sont pas confus par la lumière de soleil (qui a une plus grande intensité).

L'illumination indirecte qui a un effet plus délicat. Son intensité est plus faible que de l'illumination directe, et elle touche la surface du terrain avec un grand angle d'incidence (l'angle entre la lumière et le vecteur normal de la surface) donc sa contribution totale dans la radiance réfléchie de terrain est relativement faible, mais cette contribution est importante pour le réalisme [NK01].

3.3 Notations et terminologie de l'illumination

Dans la simulation d'un terrain Lambertian (diffus), la radiance sortante (Lo) d'un point p de terrain est donnée par :

$$Lo = \frac{C(p)}{\pi} \quad \int_{\vec{v}\in H(p)} Li(p,\vec{v})\vec{N}(p)\cdot\vec{v}\,d\Omega \tag{3.1}$$

Où $\vec{N}(p)$ est le vecteur normal à p, $H(p)$ est l' hemisphere de vecteur unitaire $\vec{v}$ sortant centré en $\vec{N}(p)$, $C(p)$ la couleur diffuse de point p de terrain, $Li(p,\vec{v})$ est la radiance incidente par la direction de $\vec{v}$ vers le point p, et $d\Omega$ est l' infinitésimal angle solide.

Les techniques discutées ici sont utilisables pour les systèmes de rendu de terrain a texture de couleur diffus à haute résolution, $C(p)$ est modulé avec des textures de basse résolution. Dans l'équation 3.1, $C(p)/\pi$ est la **BRDF** du terrain et l'intégrale est l'irradiance. $1/\pi$ est le facteur de normalisation, pour cela on va calculer l'irradiance normalisée qui est l'irradiance divisée sur π on va noter cette quantité par l pour la différentier de la radiance (L) et l'irradiance non normalisée (E). ceci est donné par :

$$l = \frac{1}{\pi} \quad \int_{\vec{v}\in H(p)} Li(p,\vec{v})\vec{N}(p)\cdot\vec{v}\,d\Omega \tag{3.2}$$

Notons que ces équations radiométriques sont continue sur le spectre électromagnétique on va simplifier ce spectre à 3 fréquences discrètes R,G et B, par conséquent toutes les quantités

de l'illumination et de couleurs qu'on utilise sont des quantités RGB.

Lumière directe du soleil On va traiter le soleil comme une simple source lumineuse directionnelle, sauf la possibilité d'occlusion partielle. Alors la contribution directe du soleil a l'irradiance dans chaque point p de terrain $(l_{SunDirect}(p))$ est donnée par :

$$l_{SunDirect}(p) = O_{sun}(p) Li_{sun} \vec{N}(p) \cdot \vec{v}_{sun} \tag{3.3}$$

Où $O_{sun}(p)$ est le facteur de visibilité du soleil au point p (égal à 1 si le soleil est complètement visible, et à 0 dans le cas où il est complètement caché), et $\vec{v}_{sun}$ est la direction de vecteur de lumière de soleil sortant.

Lumière directe du ciel Le ciel est une source lumineuse hemisphere diffuse, où l'intensité et la couleur émise varie sur l' hemisphere, pour calculer la lumière de ciel pour un point p donné de terrain, il faut intégrer toute la lumière qui vient de toutes les directions de l'ensemble $D(p)$ (le sous ensemble de $H(p)$ qui est pas caché par d'autre points de terrain).alors la contribution directe de ciel à l'irradiance dans chaque point p de terrain est donnée par :

$$l_{SkyDirect}(p) = \frac{1}{\pi} \quad \int_{\vec{v} \in D(p)} Li_{Sky}(\vec{v}) \vec{N}(p) \cdot \vec{v} \, d\Omega \tag{3.4}$$

Illumination indirecte L'illumination indirecte est le résultat de l'inter réflexion de la lumière entre les points du terrain, pour calculer cette illumination des algorithmes d'illumination globale tel que la radiosité sont nécessaires. Ces algorithmes ne peuvent s'exécuter en temps réel. On peut faire l'approximation de la lumière indirecte par des algorithme plus rapides [NK01].

3.4 Calcul analytique temps réel de l'illumination en plein air

Cette technique [NK01] basée sur des hypothèses de simplification, et d'approximations et est fondée sur l'utilisation des données pré calculées pour rechercher la solution de l' l'illumination en plein air en temps réel.

On sépare l'illumination en plein air en deux problèmes séparés : l'utilisation unique de la lumière du soleil et la lumière émise du ciel, il faut résoudre chaque problème et on doit

combiner les deux solutions pour arriver à la solution globale. Cette technique est utilisée pour mettre à jour la texture de terrain à chaque image.

Cette technique travaille avec le changement des conditions de l'illumination, et le coût nécessaire pour le pré-calcul est relativement petit [NK01].

Utilisation de la lumière du soleil uniquement : On suppose que le soleil circule en arc zénithal (arc circulaire qui passe directement en haut de son point le plus haut). Pour chaque texel de terrain on pré-calcule et on sauvegarde l'angle horizon dans le même plan comme l'arc de soleil, de cette manière on va créer une carte horizon (introduit par Max en [MAX88]).

Les angles horizon sont mesurés à un quart de cercle et prennent des valeurs entre 0 et 1, ces valeurs sont stokées sur 16 bits.

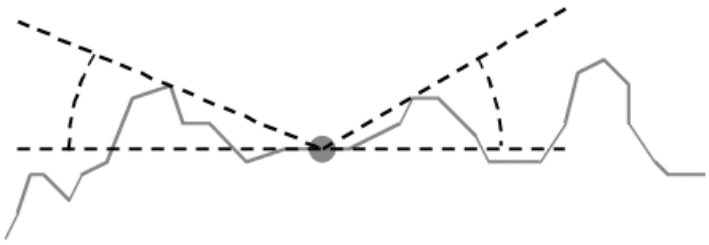


FIG. 3.1 – Angle horizon

Tout angle horizon à un point p doit être maintenus par $H(p)$; l' hémisphere de vecteur unitaire $\vec{v}$ sortant centré en normal de la surface, car l'équation de la radiosité est linéaire uniquement à l'intérieur de $H(p)$, à l'extérieur la contribution de la lumière est égale à 0. Ceci est l'horizon effectif comme le montre la figure ci dessous.

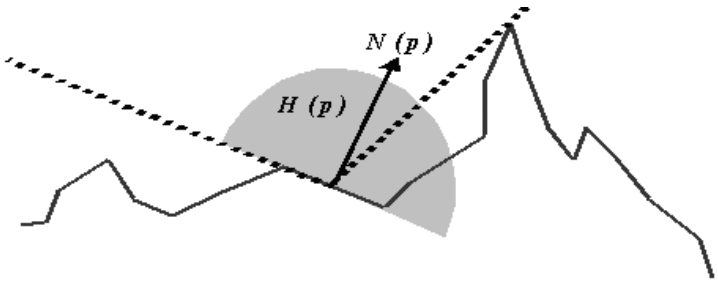


FIG. 3.2 – horizon effectif

La normale de surface est calculée à chaque texel de lightmap (carte lumière qui représente le terrain).

En temps réel quand le soleil change sa position et sa couleur il faut calculer l'angle maximum et minimum pour chaque image :

$$\theta_{Min} = \theta_{Center} - \frac{1}{2}\alpha \tag{3.5}$$

$$\theta_{Max} = \theta_{Center} + \frac{1}{2}\alpha \tag{3.6}$$

Où θ_{Center} l'angle d'élévation du centre de soleil et α est le diamètre angulaire de soleil (environ de 1/2 degrés permet de produire l'effet des ambres légères aux frontières).

Pour chaque image on calcule la RGB de l'illumination pour chaque normale par

$$Li_{sun}\vec{N}(p) \cdot \vec{v}_{sun} \tag{3.7}$$

Où Li_{sun} est la couleur courante de soleil, on stocke cela dans une table pré-calculée de l'illumination, cette table est indexée par la normale.

on parcour les texels de la carte lumière qui ont besoin d'être mis à jour, puis on cherche la contribution de la lumière de soleil à partir de la table pré-calculée par l'indexation par normal, nous on multiplie enfin par le facteur de visibilité, ce facteur sera calculé par la comparaison de l'angle maximum et l'angle minimum de soleil avec l'angle horizon. Si l'angle horizon est plus petit que l'angle minimum de soleil, alors le facteur sera 1. S'il est plus grand que l'angle maximum, le facteur sera 0. S'il est entre l'angle maximum et minimum, le facteur sera calculé par :

$$O_{sun} = \frac{\theta_{Max} - \theta_{Horizon}}{\theta_{Max} - \theta_{Min}} \tag{3.8}$$

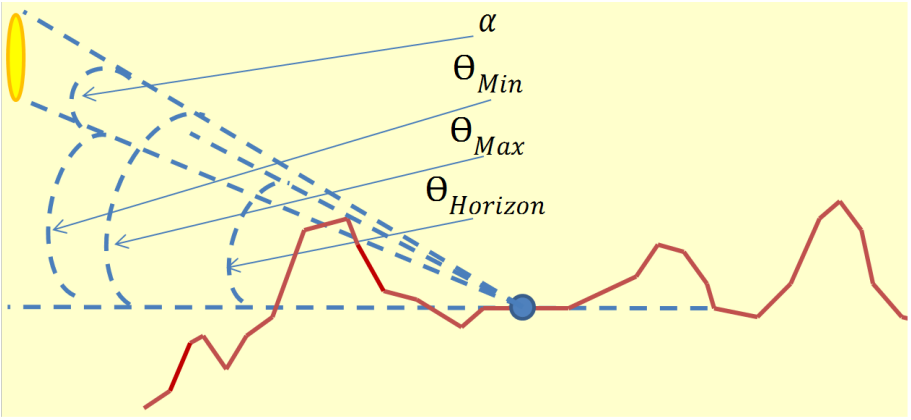


FIG. 3.3 – Visibilité du soleil

Lumière émise du ciel uniquement : Pour la lumière émise du ciel l'approche est comme suit :

- Diviser le ciel en un nombre restreint de patchs (mailles) et supposer qu'une couleur constante est affectée pour chaque patch.
- Pré-calculer pour chaque texel de lightmap et chaque patch de ciel la contribution du patch à ce texel.
- Par parcours de tous les texels de lightmap, multiplier les couleurs courantes des patch par les facteurs de contribution pré-calculés.
- Ajouter la lumière de ciel résultante à la lumière du soleil (calculée comme nous avons décrit dans la section précédente), et écrivons le résultat au lightmap.

Le pré-calcul emploie de l'équation 3.4 pour le calcul de l'illumination directe de ciel. Cette équation intègre la radiance entrante émise par les parties visibles de ciel. Cependant, aucune intégration n'est effectuée des parties cachées par le terrain. Ceci va produire des résultats foncés excessivement, car l'équation considère pour chaque point du terrain comme étant entourant le point est un noir mat parfait qui ne réfléchit aucune lumière. Dans la théorie, la solution à ce problème impliquerait de calculer la lumière réfléchie réelle du terrain entourant en utilisant un certain type de solution globale itérative d'illumination, mais la lenteur du processus de pré-calculs posent un problème.

Heureusement, Stewart et Langer [AM97] ont trouvé une bonne approximation pour l'illumination globale dans des conditions d'éclairage diffus (cette approximation, et son usage pour le rendu de terrain est également discutée par Stewart dans [STE98]). L'idée fondamentale est que les points visibles entre eux ont un éclairage semblable ; un point du terrain visible d'un point dans une vallée foncée est candidat à être également dans une vallée foncée, un point

visible d'une crête lumineuse de montagne est candidat à être également dans une crêtes lumineuse de montagne, etc.. Par conséquent, en éclairant un point du terrain, on peut supposer que tous autres points du terrain visibles à partir de ce point ont la même radiance que le point éclairé. Avec de l'algèbre, ceci représente une expression de forme fermée des inter réflexions et la lumière directe de ciel. Cette approximation a été examinée [STE98] et les erreurs résultantes ont été mesurées et se sont avérées tout à fait petites. Les papiers originaux ont assumé un terrain monochromatique constant, mais le même principe généralisé au terrain multi couleur : les points de terrain capables de voir d'autres points de terrain du même type de terrain (forêt, neige, etc. . .)ainsi ils auraient la même couleur. Les détails de petite taille de texture n'importent pas puisque notre solution d'éclairage est de résolution beaucoup inférieure que les textures de terrain. On peut exprimer mathématiquement, cette approximation donne l'équation suivante :

$$l_{Sky}(p) = \frac{1}{\pi} \quad \int_{\vec{v}\in D(p)} Li_{Sky}(\vec{v})\vec{N}(p)\cdot\vec{v}\,d\Omega + \frac{1}{\pi} \quad \int_{\vec{v}\in H(p)\setminus D(p)} Lo_{Sky}(p)\vec{N}(p)\cdot\vec{v}\,d\Omega \quad (3.9)$$

(note :la dérivation suivante est de Stewart et de Langer [AM97]). On assumera maintenant une couleur constante simple de ciel (un patch - plus tard on montre comment manipuler des patchs multiples), faisant à Li_{Sky} une constante. On substitue également $C(p)l_{Sky}(p)$ à $LoSky(p)$ (employant la couleur moyenne de texture de la région extérieure couverte par le texel de lightmap pour $C(p)$).Alors de l'algèbre simple nous donne :

$$l_{Sky}(p) = \frac{Li_{Sky}\frac{1}{\pi} \quad \int_{\vec{v}\in D(p)}\vec{N}(p)\cdot\vec{v}\,d\Omega}{1 - C(p)(1 - \frac{1}{\pi} \quad \int_{\vec{v}\in D(p)}\vec{N}(p)\cdot\vec{v}\,d\Omega)} \quad (3.10)$$

On doit calculer un facteur (on l'appellera f_{Sky}) qu'on multipliera par Li_{Sky} pour obtenir le résultat final. On peut plus loin simplifier l'expression par $K(p)$ par substitution pour les parties répétées :

$$f_{Sky}(p) = \frac{K(p)}{1 - C(p)(1 - K(p))} \quad (3.11)$$

$$K(p) = \frac{1}{\pi} \quad \int_{\vec{v}\in D(p)}\vec{N}(p)\cdot\vec{v}\,d\Omega \quad (3.12)$$

Maintenant on doit calculer $K(p)$. D'abord, on doit exprimer la surface visible $D(p)$ de ciel. pour la carte d'horizon [MAX88] au lieu de juste deux directions, maintenant on calcule les angles maintenus d'horizon dans les huit directions cardinales (N, NW, W, SW, S, SE, E,

NE). (deux de ces derniers seront également employés pour la lumière du soleil comme décrit ci-dessus). Pour exprimer $D(p)$, on divise l'horizon en huit secteurs numérotés de 0 à 7, de sorte que le secteur i englobe l'angle horizon $[(\pi/4)i, (\pi/4)(i+1)]$. On emploiera l'angle approprié d'horizon (mesuré à partir de la verticale cette fois) comme angle d'altitude ϕ_i pour chaque secteur. Basé sur ces derniers, Stewart et Langer[AM97] nous donnent une expression pour $K(p)$:

$$K(p) = \frac{1}{2\pi}\vec{N}(p) \cdot \sum_{i=0}^{7}((\phi_i - \frac{\sin 2\phi_i}{2})\Delta\sin_i \quad (\phi_i - \frac{\sin 2\phi_i}{2})\Delta\cos_i \quad \frac{\pi}{4}\sin^2\phi_i) \tag{3.13}$$

$$\Delta\sin_i = \sin(\frac{\pi}{4}(i+1)) - \sin(\frac{\pi}{4}i) \tag{3.14}$$

$$\Delta\sin_i = \cos(\frac{\pi}{4}i) - \cos(\frac{\pi}{4}(i+1)) \tag{3.15}$$

Les trois expressions à l'intérieur de la somme dans l'équation 3.13 sont les 3 composantes d'un vecteur. La somme produira un vecteur, puis le produit scalaire entre ce vecteur et $\vec{N}(p)$ est calculé et le résultat est multiplié par une constante.

On remarque également que $\Delta\sin_i$ et $\Delta\cos_i$ sont des constantes et peuvent être calculés une fois et réutilisés.

Chaque vecteur étant additionné dépend seulement ϕ_i, on pré-calcule ce vecteur pour chaque secteur et pour 256 valeurs ϕ_i ayant pour résultat une table $256 * 8$. Noter qu'une meilleure exactitude est réalisée si on emploie la tangente de l'angle pour la consultation de table (la tangente de l'angle est calculée facilement quand l'angle horizon est calculé).

Après que $K(p)$ soit calculé, la procedure est la suivante :

- Calculer le facteur de lumière de ciel (une valeur de RVB) et le stocker pour chaque texel de lightmap.
- Multiplier le facteur de lumière de ciel avec la couleur courante de ciel chaque fois pour obtenir la contribution de lumière de ciel.
- Dans le cas des patchs multiples, le facteur de lumière de ciel doit être calculé et stocké pour chaque patch, la couleur de patch doit être multipliée par le facteur de chaque patch et additionner les résultats pour obtenir la contribution totale de lumière de ciel [NK01].

3.5 Méthodes de calcul d'ombres pour les terrains

Réussir à afficher le terrain est une bonne chose. Mais il manque une chose primordiale : des ombres pour bien montrer le relief. Dans cette partie nous allons présenter trois techniques pour le calcul de l'éclairage [CLI95].

L'ajout d'ombres dans la génération de la texture ajoute un effet de réalisme important au rendu. L'image suivante compare trois rendus différents.

A gauche, la version la plus basique mais qui permet d'avoir un résultat quasi-identique au deuxième (celui du milieu) qui utilise un produit scalaire. Ces deux techniques sont acceptables mais elles ne font qu'un calcul localisé. Si on a donc une montagne, on n'aura pas l'effet d'ombre que la montagne va faire sur la vallée d'à côté. C'est le dernier algorithme qui permet en plus de faire ressortir un plus grand réalisme en ce qui concerne les ombres.

FIG. 3.4 – Comparaison de calculs d'ombre

3.5.1 Méthode de coefficient par différence entre les hauteurs

La technique la plus simple dans le calcul d'ombre, consiste à utiliser un vecteur lumineux, de l'employer comme base de tous les algorithmes qui vont suivre, et de considérer que ce vecteur est constant pour tout les point du terrain, ceci car le soleil est trés distant.

Pour chaque pixel on va calculer le coefficient de luminosité. Ce sera un nombre flottant entre 0 et 1 qu'on multiplie par la couleur sans calcul d'ombre pour obtenir la couleur finale, l'ensemble des coefficients est une lightmap (carte lumière qui représente le terrain)[CLI95].

Calcul d'ombre

$\rightarrow /* \; Calcul\; d'ombre \; */$

$\rightarrow r *= \; diff;$

$\rightarrow g *= diff;$

$\rightarrow b *= diff;$

Pour calculer ce coefficient, on a fixer la direction du rayon du soleil dans une structure et on définera ce vecteur comme étant $(l\_veci,\ l\_vecj)$ (donc le rayon du soleil possède une direction $(l\_veci,\ pl\_vecj)$), par conséquent, pour un pixel (i, j), le pixel $(i - l\_veci, j - l\_vecj)$ est plus élevé alors la luminosité du pixel (i, j) sera amoindrie (en effet, le pixel $(i - l\_veci, j - l\_vecj)$ cachera un peu le pixel (i, j)).

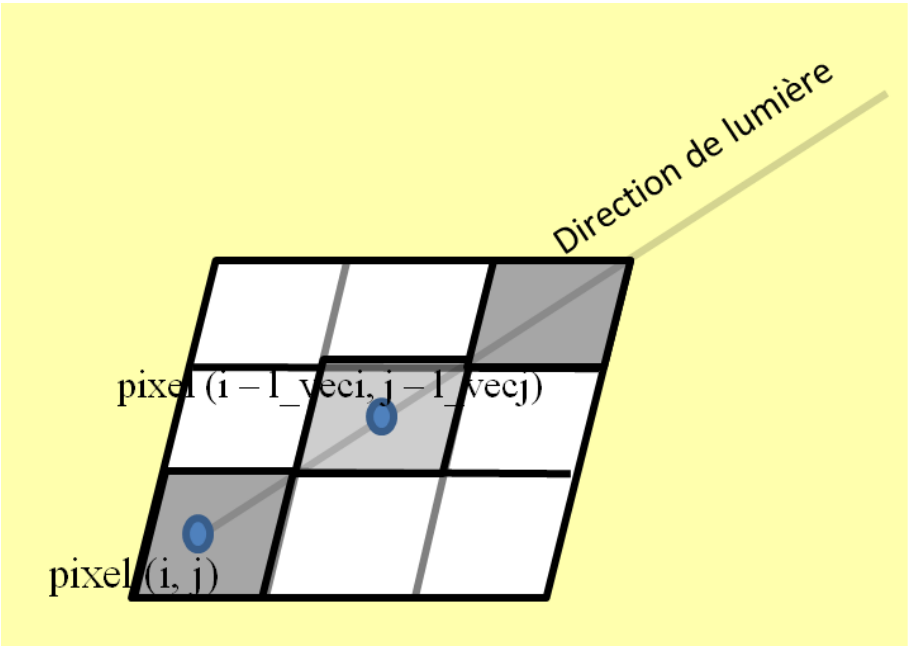


FIG. 3.5 – Suivie de vecteur lumière

En utilisant cette idée, on peut définir donc une première fonction $Terrain\_calcLightMap\_Simple$ qui sera donné par :

Calcul d'ombre :

$\rightarrow double\ Terrain\_calcLightMap\_Simple(SDL\_Surface *image, int\ i,\ int\ j,\ int\ maxi,\ int\ maxj)$

$\rightarrow \{$

$\rightarrow \quad if((i - l\_veci >= 0)\&\&(i - l\_veci < maxi)\&\&(j - l\_vecj >= 0)\&\&(j -$

$\rightarrow \quad l\_vecj < maxj))$

$\rightarrow \quad\quad return\ 1.0 - (Terrain\_GetHauteur(image, i - l\_veci, j - l\_vecj, maxi, maxj)$

$\rightarrow \quad\quad\quad - Terrain\_GetHauteur(image, i, j, maxi, maxj))/l\_adouc;$

$\rightarrow \quad else$

$\rightarrow \quad\quad return\ 1.0f;$

$\rightarrow\}$

Comme on le remarque, on utilise la différence entre les deux hauteurs pour calculer le taux d'ombre. Si cette différence est grande, alors le résultat sera proche de 0, dans le cas contraire, le résultat sera proche de 1.

La valeur l_adouc permet de moduler la quantité d'ombre créée. Si ce nombre est petit, les zones seront très ombragées.

3.5.2 Limitations de la première méthode

La première méthode possède l'inconvénient d'être trop localisée, ce qui provoque des imperfections dans le calcul des ombres.

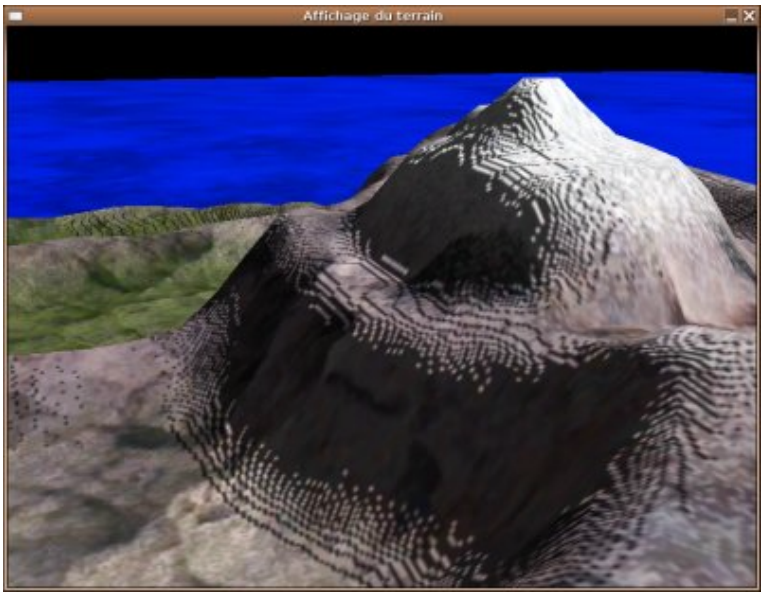


FIG. 3.6 – Défauts de la première méthode

Pour remédier à ce problème, au lieu d'avoir un calcul localisé, pour le calcul de l'ombre du pixel (i, j), on va diviser la carte lumière qui représente le terrain en un ensemble de patchs et utiliser la moyenne du coefficient d'ombre de chaque pixel avoisinant aussi (même patch). Ceci permettra d'éviter ce genre de défaut.

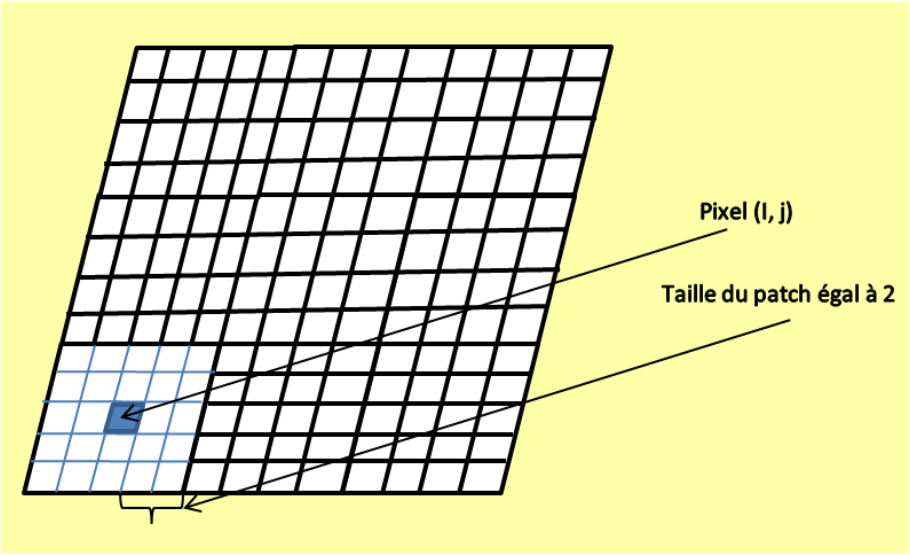


FIG. 3.7 – Patch de taille égal à 2

Puisque cette idée de faire une moyenne va revenir par la suite. On va donc faire une fonction qui applique ce calcul de moyenne [CLI95]. Il faudra donc faire deux passes.

En effet, pour faire correctement les choses, on stocke dans un tableau assez grand (qu'on appellera lightmap) les coefficents d'ombre de chaque pixel. Ensuite, à l'aide d'un deuxième tableau temporaire initialisé par des zéros, on procédera au calcul de chaque coefficient. Le pseudo-code suivant représente la fonction qui fait le calcul des moyennes :

Calcul d'ombre Version 2 :

```
→   /* Compteur pour savoir combien d'éléments ont été sommés */
→   cnt = 0;

→   for(k = i − taille_patch; k <= i + taille_patch; k++)
→   {
→       for(l = j − taille_patch; l <= j + taille_patch; l++)
→       {
→           /* Si les coordonnées sont bonnes */
→           if((k >= 0)&&(l >= 0)&&(k < maxi)&&(l < maxj))
→           {
→           tmplightmap[i][j]+ = lightmap[k][l];
→           cnt++;
→           }
```

```
→                     }
→                 }

→                 /* Calcul de la moyenne */
→                     tmplightmap[i][j] /= cnt;
```

Et voici le résultat :

FIG. 3.8 – Première méthode après l'application du filtre

Le paramètre *taille_patch* est utiliser pour définir la grandeur du filtre, plus il augmente, plus il y aura de calculs pour terminer la génération. De plus, d'un point de vue rendu, cela n'est pas forcément une bonne idée donc on a mis un filtre $4 * 4$ (donc le paramètre taille_patch vaut 4).

3.5.3 Méthode de produit scalaire : L dot N

L'avantage de la méthode précédente est la facilité d'utilisation. Mais elle demeure une approximation de la vraie solution qui consiste à utiliser un produit scalaire entre la normal du point et le vecteur lumière [CLI95].

En effet, c'est le signe de ce produit scalaire qui nous orientera sur la façon avec la quelle le pixel devrait être ombragé. Si le signe est positif, alors la normale et le vecteur lumière sont dans les mêmes directions. La lumière ne peut donc pas illuminer ce point.

Sinon, le produit est négatif, auquel cas, on prendera l'opposé de la valeur du produit scalaire comme coefficient de luminosité. On peut donc définir une nouvelle fonction *Terrain_calcLightMap_LDOTN*

$\rightarrow$ *double Terrain_calcLightMap_LDOTN(SDL_Surface *image, int i, int j, int maxi, int maxj);*

Cette fonction vérifie si nous on ne va pas provoquer un débordement mémoire. Ensuite, elle calcule une normale.

On utilise en fait les positions des points voisins. En effet, en utilisant les points $(i-1, j-1)$, $(i+1, j-1)$ et $(i+1, j+1)$ on définit un triangle. On pourra donc calculer un vecteur normal à partir de ces trois points.

Début du calcul L . N :

$\rightarrow$ */* Complétons ces vecteurs*

$\rightarrow$ ** Premier vecteur sera le vecteur* $(i-1, j-1)$ *vers* $(i+1, j-1)$

$\rightarrow$ ** Deuxième vecteur sera le vecteur* $(i+1, j-1)$ *vers* $(i+1, j+1)$

$\rightarrow$ **/*

$\rightarrow v1.x = 2; v1.y = 0;$

$\rightarrow v1.z = Terrain\_GetHauteur(image, i+1, j-1, maxi, maxj)$

$\rightarrow \quad\quad -Terrain\_GetHauteur(image, i-1, j-1, maxi, maxj);$

$\rightarrow v2.x = 0; v2.y = 2;$

$\rightarrow v2.z = Terrain\_GetHauteur(image, i+1, j+1, maxi, maxj)$

$\rightarrow \quad\quad -Terrain\_GetHauteur(image, i+1, j-1, maxi, maxj);$

$\rightarrow$ */* Normalise * /*

$\rightarrow Vecteur\_Normalise(\&v1);$

$\rightarrow Vecteur\_Normalise(\&v2);$

$\rightarrow$ */* On cherche la normale * /*

$\rightarrow Vecteur\_pVect(\&v1, \&v2, \&n);$

$\rightarrow$ */* On normalise la normale * /*

$\rightarrow Vecteur\_Normalise(\&n);$

En regardant le signe de z, on connaitre le sens de l'orientation du vecteur vers le haut ou vers le bas.

Changement de sens pour le vecteur normal :

```
→ /* On vérifie que le vecteur est dans le bon sens */
→ if(n.z < 0)
→       {
→       n.x* = −1;
→       n.y* = −1;
→       n.z* = −1;
→       }
```

On utilise toujours des vecteurs normaux, aux quels on intègre des facteurs d'ajustement par la suite. Comme c'est fait ici :

Fin de la fonction :

```
→ /* On a la normale , on calcul maintenant L dot N */
→ tmp = Vecteur_ProdScal(&light, &n);

→ if(tmp < 0)
→       return − param.l_ldotnmultiple * tmp;
→ else
→       return0.0f;
```

On voit bien que si tmp est négatif alors le pixel sera illuminé. Vu que on veut une valeur entre 0 et 1, on multiplie la valeur par un ajustement. Cet ajustement dépendra du terrain et du vecteur lumière. Comme pour la première méthode, il sera plus sage d'appliquer un filtre comme pour la partie précédente sur le calcul d'ombre.

Voici une image illustrant cette solution :

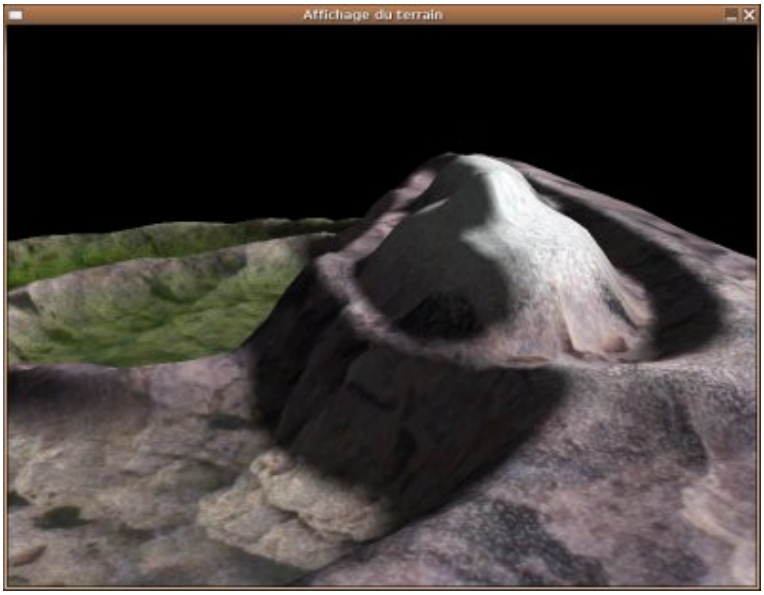


FIG. 3.9 – Deuxième méthode (L dot N) utilisant un filtre

3.5.4 Méthode de lanceur de rayon

La technique de lanceur de rayon [CLI95, STE98] n'est qu'une extension de la dernière. En effet, on va utiliser le produit scalaire lorsqu'un calcul de coefficient sera nécessaire. La problématique de cette solution est donc de savoir quand est-ce qu'un tel calcul est nécessaire.

On part toujours de la même hypothèse : on a un vecteur lumière connu. Pour simplifier la méthode, on suppose que le soleil, étant tellement loin, possède le même vecteur rayon lumineux pour tous les points du terrain [CLI95].

S'il existe une intersection entre le vecteur lumière partant du point qui nous intéresse et le reste du terrain, alors il ne peut pas avoir de rayon arrivant ici. On mettra donc ce point à 0 ; S'il n'y a pas d'intersection, alors on va devoir utiliser la technique du produit scalaire [CLI95, STE98].

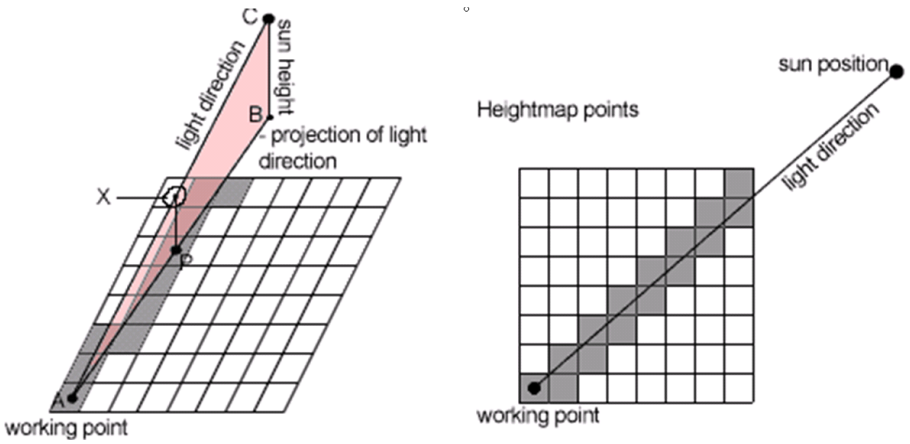


FIG. 3.10 – principe lanceur de rayon

Finalement, cette solution, bien que gourmande en ressources (vu le nombre de calculs à faire) n'est pas très difficile à mettre en oeuvre. Voici son implémentation :

Lanceur de rayon pour le coefficient de luminosité :

$\rightarrow$ *double Terrain_calcLightMap_Ray(SDL_Surface * image,int i, int j, int maxi,*
*int maxj, SVecteur * light)*
$\rightarrow$ {
$\rightarrow$ *SPoint cur;*
$\rightarrow$ *float tmp;*

$\rightarrow$ */ **
$\rightarrow$ **Verification s'il y a une intersection avec le terrain, si*
$\rightarrow$ ** c'est le cas, il n'y aura pas de lumiere ici*
$\rightarrow$ **/*

$\rightarrow$ */ **
$\rightarrow$ ** Point de depart, le point(i, j) avec son hauteur*
$\rightarrow$ **/*
$\rightarrow$ *tmp = i;*
$\rightarrow$ *tmp/ = maxi;*
$\rightarrow$ *tmp* = image− > h;*

$\rightarrow$ *cur.x = tmp;*

```
→       tmp = j;
→       tmp/ = maxj;
→       tmp* = image− > w;

→       cur.y = tmp;
→       cur.z = Terrain_GetHauteur(image, i, j, maxi, maxj);
```

On a le point de départ avec son hauteur. On a de nouveau fait un changement de repére entre la position dans la texture à générer et l'image de niveaux. Ensuite vient le calcul pour savoir s'il existe une intersection ou non.

Recherche d'intersection :

```
→       /* Tant qu'on est dans le terrain */
→       while((cur.x >= 0)&&(cur.y >= 0)&&(cur.x < image− > h)&&
→           (cur.y < image− > w))
→           {
→           /* On recupere la hauteur courante */
→           tmp = Terrain_GetHauteur(image, (int)cur.x, (int)cur.y, image− >
→           h, image− > w);

→           /* Si c'est au − dessus du maximum possible du terrain */
→           if(cur.z > param.hautmax)
→               {
→               /* On sort, pas d'intersection possible */
→               break;
→               }

→         /* Si le terrain est au − dessus du vecteur de lumiere, le terrain cache la lumiere *
/
→           if(tmp > cur.z)
→               {
→               return 0.0;
→               }
```

```
→            /* Sinon, on continue avec le vecteur de lumiere */
→            cur.x+ = light− > x;
→            cur.y+ = light− > y;
→            cur.z+ = light− > z;
→           }

→       /* Sinon pas d'intersection , on calcul la luminosité avec le calcul L dot N */
→       return Terrain_calcLightMap_LDOTN(image, i, j, maxi, maxj);
→}
```

Si le niveau du terrain est au-dessus du vecteur allant vers la lumière alors il y a une intersection. Sinon on peut calculer le coefficient de luminosité utilisant le produit scalaire de la deuxième méthode. Voici une image montrant ce calcul d'ombre :

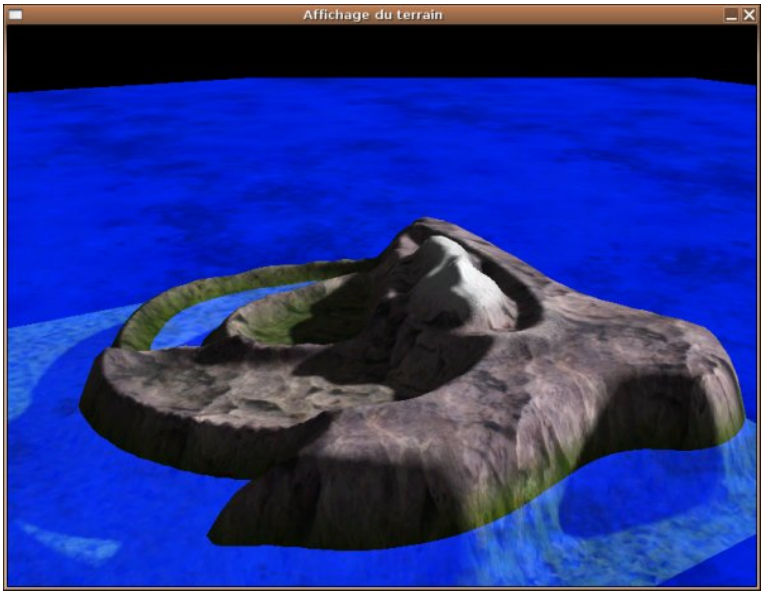


FIG. 3.11 – Méthode du lanceur de rayon

3.6 Bilan

Aprés avoir étudié les principales méthodes utilisées pour l'illumination et le calcul des ombres pour les terrains, nous pouvons dresser le bilan :

Méthode	Caractéristiques	Avantages	inconvénients
Par différence entre les hauteurs	– Basée sur la différence des hauteurs des pixels voisins.	– Facile à implémenter. – Très rapide	– Méthode trop localisé, ce qui provoque des imperfections dans le calcul des ombres.
Méthode de produit scalaire	– Basée sur le produit scalaire entre la normale du point et le vecteur lumière.	– Plus exacte que la méthode précédente.	– Méthode moins rapide.
Méthode de lanceur de rayon	– basée sur la suivie de vecteur lumière partant du point.	– Solution fiable	– Relativement lente. – Ne prend pas en considération le diamètre de soleil.
Calcul analytique	– Basée sur les données pré-calculées : carte horizon, facteurs de contribution.	– Rapide et réaliste. – Prise en charge de diamètre angulaire de soleil.	– Phase d'initialisation (pré-calcul de donnée) couteuse.

3.7 Conclusion

Nous avons présenté une techniques d'illumination temps réel du terrain. Cette technique est le calcul analytique temps réel de l'illumination en plein air [NK01] ; elle optimise le rendu par les hypothèses de simplification et les données pré-calculées : la carte horizon, les facteurs de contribution.

Nous avons aussi présenté trois techniques pour le calcul des ombres [CLI95] qui ajoutent un effet de réalisme important au rendu. L'avantage de la première méthode est la facilité et la simplicité. Mais elle demeure une approximation de la vraie solution qui consiste à utiliser un produit scalaire entre la normale du point et le vecteur lumière. La dernière méthode n'est qu'une extension de la deuxième. En effet, nous allons utiliser le produit scalaire lorsqu'un calcul de coefficient sera nécessaire ; si le point est caché par d'autres points du terrain, il n'est pas nécessaire de calculer le produit scalaire.

Mais il faut prendre en considération que le soleil n'est pas un point, il possède un diamètre angulaire. Nous allons essayer de profiter d'une idée présentée dans la section 3.4 et l'exploiter pour le calcul de l'ombre, pour ne pas compliquer les choses nous allons introduire d'autres critères de simplification.

Deuxième partie

Contributions

Chapitre 4

Contributions

4.1 Introduction

D'aprés l'état de l'art que nous avons présenté dans la 1^{iere} partie de ce mémoire, le sujet de génération de terrains possède trois axes de recherche principaux : la modélisation, la génération de textures, et l'illumination. Dans chaque axe les chercheurs inventent toujours de nouvelles méthodes pour améliorer et régler les problèmes des méthodes précédemment inventées, et les recherches sont toujours ouverts dans les trois axes.

Notre approche se situe dans ce contexte, elle consiste à se concentrer sur l'illumination de terrain afin de bien montrer le relief d'une part et d'ajouter un effet de réalisme important au rendu d'autre part.
Nous allons aussi introduire quelques propositions et contributions afin d'optimiser la phase de modélisation des terrains (réduire le coût de calcul).

4.2 Motivations pour de nouvelles approches

Les techniques d'illumination globale sont les plus réalistes pour le rendu, mais elles restent moins intéractives.

Une méthode présentée dans la section 3.4, sert à optimiser le rendu par les hypothèses de simplification et les données précalculées : la carte horizon, les facteurs de contribution.

Les techniques de calcul d'ombre vues dans l'état de l'art sont sont simples, mais elles manquent de réalisme. En effet, nous remarquons qu'elles ne prennent pas en considération le fait que le soleil n'est pas une source lumineuse ponctuelle, ceci est traité par la méthode

présetée dans la section 3.4, mais la solution nécessite une phase trés complexe de précalcul des angles horizon et des facteurs de contribution, et considère que la modélisation de terrain est préétablie, bien qu'il est nécessaire quelques fois de chercher à modéliser et à faire le rendu en même temps et de façon interactive (temps réel).

Nous avons vue que l'algorithme de Perlin est efficace et donne des terrains réalistes, mais il est lourd en terme de calculs à cause du nombre important d'interpolation pour chaque calque.

Des nouvelles techniques sont donc nécessaires pour prendre en charge ces points.

4.3 Approche proposée pour la modélisation

4.3.1 Principe

Nous avons vu que la technique de Diamond-Square est simple et rapide, mais elle possède le problème des petites crêtes (sommets).

La méthode de Perlin donne de bons résultats, mais l'utilisation d'une suite de calques dans les quels on interpole entre les valeurs, rend l'algorithme très lourd et plein de calculs.

Nous proposons une alternative qui consiste à générer un terrain par l'algorithme de Diamond-Square, en ajoutant au résultat un calque de Perlin. La problématique de cette solution est donc de savoir, quels paramètres nous allons choisir pour ce calque.

Si on utilise un calque à une grande fréquence, les points seront très proches entre eux, donc il faut choisir une faible persistance (pondération) pour ne pas produire des incohérences dans le terrain, mais cette persistance ne sera pas suffisante pour influencer sur les crêtes afin de les éliminer.

Dans le cas où on augmente la persistance il faut aussi réduire la fréquence pour éviter les incohérences, cette réduction de fréquence implique que le calque affecte la forme générale du terrain et non les détails.

Nous allons donc adopter le processus d'essai afin de déterminer les bons paramètres, mais pour l'interactivité il faut fixer les paramètres dés le départ par le choix des valeurs moyennes pour assurer des résultats acceptables.

Nous allons commencer par la génération d'un terrain par la méthode Diamond-Square, en utilisant les fonctions :

/* Generation d'un nombre aleatoire */

$\rightarrow float\ \ Randomize()$

```
→ {
→    return (float)rand()/RAND_MAX;
→}
/* Phase du "diamant" */
→ float DiamondStep(unsigned int unI, unsigned int unJ, unsigned int unHalfSpacing)
→ {
→  float sum = 0.0;
→  int n = 0;

→  if((unI >= unHalfSpacing)&&(unJ >= unHalfSpacing))
→  {
→     sum+ = param.m_vectPoints[unI − unHalfSpacing][unJ − unHalfSpacing];
→     n + +;
→  }

→  if((unI >= unHalfSpacing)&&(unJ + unHalfSpacing < m_unSize))
→  {
→     sum+ = param.m_vectPoints[unI − unHalfSpacing][unJ + unHalfSpacing];
→     n + +;
→  }

→  if((unI + unHalfSpacing < m_unSize)&&(unJ >= unHalfSpacing))
→  {
→     sum+ = param.m_vectPoints[unI + unHalfSpacing][unJ − unHalfSpacing];
→     n + +;
→  }

→  if((unI + unHalfSpacing < m_unSize)&&(unJ + unHalfSpacing < m_unSize))
→  {
→     sum+ = param.m_vectPoints[unI + unHalfSpacing][unJ + unHalfSpacing];
→     n + +;
→  }
```

```
→  return sum/n;
→}
/* Phase du "carre" */
→ float SquareStep(unsigned int unI, unsigned int unJ, unsigned int unHalfSpacing)
→ {
→  float sum = 0.0;
→  int n = 0;

→  if(unI >= unHalfSpacing)
→  {
→     sum+ = param.m_vectPoints[unI − unHalfSpacing][unJ];
→     n + +;
→    }

→  if(unI + unHalfSpacing < m_unSize)
→  {
→     sum+ = param.m_vectPoints[unI + unHalfSpacing][unJ];
→     n + +;
→ }

→  if(unJ >= unHalfSpacing)
→  {
→     sum+ = param.m_vectPoints[unI][unJ − unHalfSpacing];
→     n + +;
→ }

→  if(unJ + unHalfSpacing < m_unSize)
→  {
→     sum+ = param.m_vectPoints[unI][unJ + unHalfSpacing];
→     n + +;
→ }
```

```
→  return sum/n;
→}
```

La fonction qui permet de générer le module du Diamond-Square ; est donnée par le pseudo-code :

```
→ void Generate(float fLeftBottom, float fLeftTop, float fRightTop, float fRightBottom)
→ {
→  param.m_vectPoints[0][0] = fLeftBottom;
→  param.m_vectPoints[0][m_unSize − 1] = fRightBottom;
→  param.m_vectPoints[m_unSize − 1][0] = fLeftTop;
→  param.m_vectPoints[m_unSize − 1][m_unSize − 1] = fRightTop;

→  //Initialisation de l'espace entre les valeurs
→  unsigned int unSpacing = m_unSize − 1;
→  unsigned int unI, unJ;

→  while (unSpacing > 1)
→  {

→    int unHalfSpacing = unSpacing/2;

→    //effectuer l'étape du diamond
→    for(unI = unHalfSpacing; unI < m_unSize; unI+ = unSpacing)
→    {
→      for(unJ = unHalfSpacing; unJ < m_unSize; unJ+ = unSpacing)
→      {
→        param.m_vectPoints[unI][unJ] = DiamondStep(unI, unJ, unHalfSpacing) +
→            Randomize() ∗ unSpacing ∗ param.m_fVariability;
→      }
→    }
```

```
→     //effectuer l'étape du carré
→     for(unI = 0; unI < m_unSize; unI+ = unHalfSpacing)
→     {
→       unsigned int unJStart = ((unI/unHalfSpacing)%2 == 0)?unHalfSpacing :
0;

→       for(unJ = unJStart; unJ < m_unSize; unJ+ = unSpacing)
→       {
→         param.m_vectPoints[unI][unJ] = SquareStep(unI, unJ, unHalfSpacing) +
→           Randomize() * unSpacing * param.m_fVariability;
→       }
→     }

→     //Diviser la carte utilisée en 2
→     unSpacing = unHalfSpacing;

→ }

→   //Déterminer la hauteur maximale de la carte
→   float m_fMax = param.m_vectPoints[0][0];

→   for(unI = 0; unI < m_unSize; unI++)
→   for(unJ = 0; unJ < m_unSize; unJ++)
→     if(param.m_vectPoints[unI][unJ] > m_fMax)
→     {
→       m_fMax = param.m_vectPoints[unI][unJ];
→     }

→   //Normalisation

→   for(unI = 0; unI < m_unSize; unI++)
```

$\rightarrow \quad for(unJ = 0; unJ < m\_unSize; unJ++)$

$\rightarrow \quad \{$

$\rightarrow \quad param.m\_vectPoints[unI][unJ] = (param.m\_vectPoints[unI][unJ] * 256)/m\_fMax;$

$\rightarrow \quad \}$

$\rightarrow \}$

Nous aurons comme résultat la matrice $param.m\_vectPoints[\,][\,]$ qui représente le modèle Diamond-Square. Après le calcul d'un ou plusieurs calques de Perlin nous effectuons le travail suivant :

/* ajout des calques successifs */

$\rightarrow \quad for(n = 0; n < octaves; n++)$

$\rightarrow \quad c->v[i][j] += mes\_calques[n]->v[i][j] * mes\_calques[n]->persistance;$

/* normalisation */

$\rightarrow \quad c->v[i][j] = (c->v[i][j] + param.m\_vectPoints[i][j])/(sum\_persistances + 1);$

Nous avons considéré que le modèle Diamond-Square est un calque de Perlin qui a une persistance égale à 1.

Voici une image résultat illustrant cette solution :

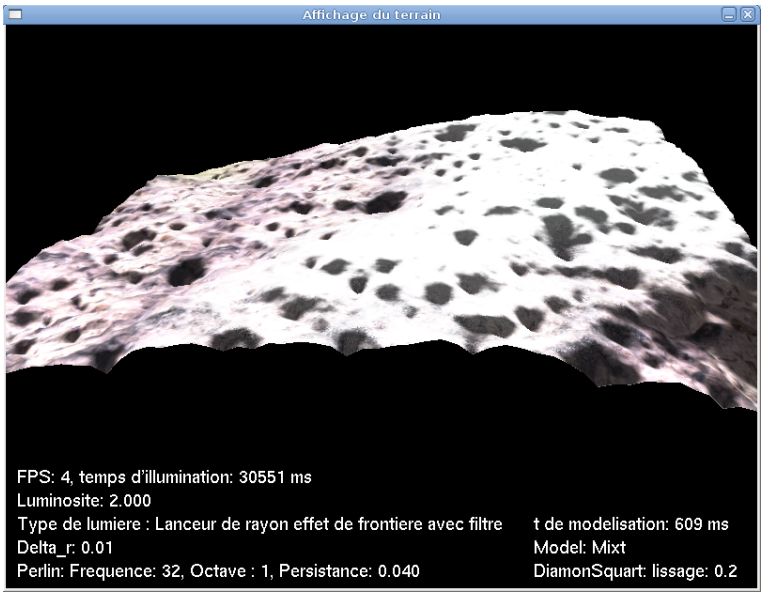


FIG. 4.1 – Modèle mixte

Si nous comparons cette image avec la solution obtenue par l'algorithme Diamond-Square classique données par la figure suivante :

FIG. 4.2 – Modèl Diamond-Square

Nous remarquons que les petites crêtes sont réduites par les détails ajoutés par le calque de Perlin, nous allons voir que cette technique est aussi plus rapide que celle de Perlin.

4.3.2 Amélioration de la technique proposée

Dans la technique précédente nous avons décomposé le terrain en deux composantes :

– Une composante qui représente la forme globale du terrain générée par la technique Diamond-Square.
– Une 2^{ieme} composante qui représente les détails du terrain générée par la technique de Perlin.

Supposons que nous avons un terrain trés large, si nous calculons la composante qui représente les détails de tout le terrain par la technique de Perlin, elle sera trés couteuse à cause du nombre important d'interpolation à calculer.

Afin d'augmenter la vitesse, nous proposons une solution qui consiste à suivre les étapes suivantes :

– Générer la composante qui représente la forme globale du terrain par la technique Diamond-Square.
– Subdiviser le terrain en un ensemble de sous parties plus petites.
– Générer une seule composante qui possède la taille d'une seule sous partie par la technique de Perlin pour représenter les détails du terrain.

– Pour chaque sous partie de la composante qui représente la forme globale, nous appliquons la même composante de détails générée dans l'étape précédente.

Nous remarquons que nous avons moins d'interpolation à calculer, alors que un temps de calcul pour la génération est plus petit.

4.3.3 Amélioration de l'approche proposée par la technique de Perlin

Pour améliorer la technique proposée par l'augmentation de la qualité et la réduction du temps de calcul, nous décomposons le terrain en deux composantes comme pour la technique précédente :

– Une composante qui représente la forme globale du terrain.
– Une 2^{ieme} composante qui représente les détails du terrain.

Nous allons donc suivre les étapes suivantes :

– Générer la composante qui représente la forme globale du terrain par un calque de Perlin à faible fréquence.
– Subdiviser le terrain en un ensemble de sous parties plus petites
– Générer une seule composante qui possède la taille d'une seule sous partie par la technique de Perlin pour représenter les détails du terrain.
– Pour chaque sous partie de la composante qui représente la forme globale, appliquer la même composante de détails générée dans l'étape précédente.

Dans la technique de Perlin les calques fins qui représentent les détails sont les plus couteux pour cela nous avons généré une seule petite partie que nous avons appliqué au reste.

Théoriquement cette amélioration permet d'obtenir exactement un terrain de Perlin avec une vitesse trés rapide que celle de la technique de Perlin.

4.3.4 Le problème des deux techniques améliorées et sa résolution

Le fait de traiter chaque sous partie indépendamment des autres, va certainement produire des discontinuités au niveau des frontières entre les différentes parties.
Pour résoudre ce problème le lissage est nécessaire, il s'agit d'interpoler entre les valeurs des frontières.

Supposons que nous avons plus d'une couche de calques qui représentent la composante des détails, nous aurons deux possibilités pour interpoler entre les valeurs des frontières selon la distance choisie entre les points :

– Une distance selon le calque le plus faible ;

– Une distance selon le calque le plus fin.

Pour le 1^{er} cas ; nous aurons des surfaces trés lisses et moins détaillées dans les zones de raccordement, mais pour le $2^{ième}$ cas ; le fait que les points sont très proches entre eux induit à des incohérences au niveau des zones de raccordement.

La solution : L'ajout de la composante qui représente les détails, mais couche par couche ; l'ajout du premier calque à toute la surface de la composante globale partie par partie, nous passons au calque suivant pour ajouter une autre couche plus détaillée, jusqu'à ce que ne reste pas de calque.

nous interpolons entre les frontières des calques au niveau de chaque couches, de cette façon nous allons **affiner le lissage exactement telque l'affinement de notre terrain**, alors nous aurons des zones de raccordement qui ont les même caractéristiques que le terrain.

4.4 Contributions pour l'illumination du terrain

Le soleil n'est pas une source lumineuse ponctuelle, il existe alors des points dans le terrain pour les quelles le soleil est partiellement visible, l'éclairage de ces points doit donc être multiplié par un facteur de visibilité de soleil. Le problème, par conséquent réside dans le calcul de ce facteur.

4.4.1 Une première version de l'approche proposée

L'approche que nous allons essayer de proposer est basée sur les méthodes présentées dans la section 3.5.

Nous allons simuler l'effet des ombres légères au frontières, posé par le diamètre angulaire de soleil. Cette technique est basée sur la dernière méthode de calcul des ombres présentée dans la section 3.5.4.

Après avoir multiplier la couleur par le coefficient de luminosité, nous allons multiplier le résultat par un facteur de visibilité V. Le problème réside donc dans le calcul de ce facteur.
Nous considérons que le vecteur lumière est un cylindre qui possède un diamètre perpendiculaire $(R = 2r)$. Le facteur de visibilité doit approximer le rapport entre la surface visible et la surface totale du vecteur lumière, donc le principe de la méthode sera comme suit :

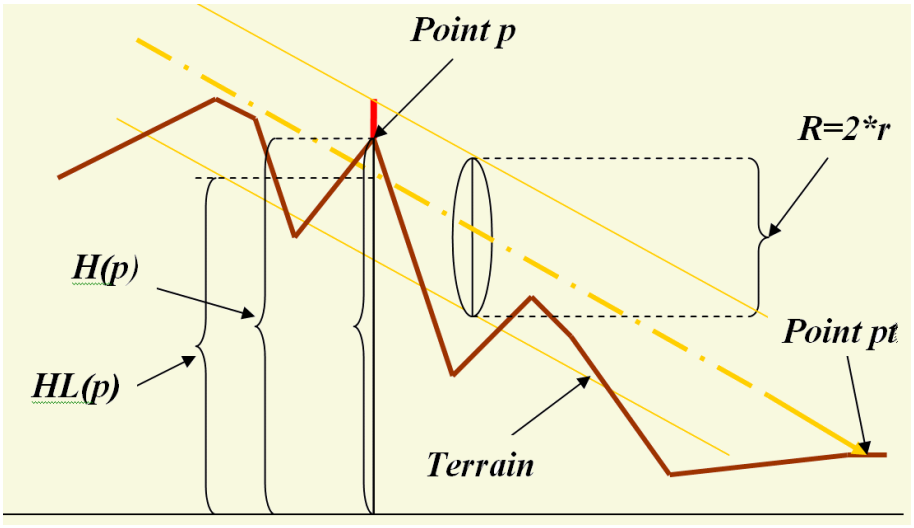


FIG. 4.3 – Principe de cylindre

Du point pt qui nous intéresse nous allons suivre le vecteur lumière, s'il n'existe pas un point p qui possède l'hauteur $H(p)$ tel que $H(p) > (HL(p) - r)$.
Où $HL(p)$ est la hauteur du centre de vecteur lumière à ce point ;
Ceci signifie que aucun point du terrain que cache une portion de la surface du vecteur lumière, donc le soleil est complètement visible, alors le facteur de visibilité du soleil sera égal à 1 $(V(pt) = 1)$ et nous allons obtenir la couleur finale :

$$CouleurFin = Couleur * (\vec{L} \cdot \vec{N}(pt)) \tag{4.1}$$

Où $\vec{N}(pt)$ est la normale au point pt et $\vec{L}$ le vecteur lumière.

S'ils existent des points qui vérifient la condition $H(p) > (HL(p) - r)$ nous aurons deux cas possibles :

- Si $H(P) \geq (HL(p) + r)$, alors le soleil est invisible $V(pt) = 0$, donc $CouleurFin = 0$.
- Si $(HL(p) - r) < H(p) < (HL(p) + r)$, alors le soleil est partiellement visible donc :

$$CouleurFin = V(pt) * Couleur * (\vec{L} \cdot \vec{N}(pt)) \tag{4.2}$$

Le facteur de visibilité est le rapport de la surface visible(longueur de la ligne rouge) sur la surface totale de vecteur lumière représentée par R, alors :

$$V(pt) = \frac{(HL(p) + r) - H(p)}{R} \tag{4.3}$$

Si nous avons plusieurs points pour les quels le soleil est partiellement visible nous gardons le facteur de visibilité minimal.

La fonction qui réalise ce travail est donnée par le pseudo-code suivant :

```
→ double Terrain_calcLightMap_RayFrontTemp(SDL_Surface *image,int i, int j, int maxi,
int maxj, SVecteur *light)
→ {
→       SPoint cur;
→       float tmp;
→       double Vsun,newVsun;

→       /*
→       *Verification s'il y a une intersection avec le terrain, si
→       * c'est le cas, il n'y aura pas de lumière
→       */

→       /*
→       * Point de depart, le point(i,j) avec son hauteur
→       */
→       tmp = i;
→       tmp/ = maxi;
→       tmp* = image- > h;

→       cur.x = tmp;

→       tmp = j;
→       tmp/ = maxj;
→       tmp* = image- > w;

→       cur.y = tmp;
→       cur.z = Terrain_GetHauteur(image,i,j,maxi,maxj);
→       Vsun = 1;
```

```
→   newVsun = 2;
→   double L_DIAM_RAYON = 0.5;

→   /* Tant qu'on est dans le terrain */
→   while((cur.x >= 0)&&(cur.y >= 0)&&(cur.x < image->h)&&
→       (cur.y < image->w))
→       {
→       /* On recupere la hauteur courante */
→       tmp = Terrain_GetHauteur(image,(int)cur.x,(int)cur.y,image->
→       h,image->w);

→       /* Si c'est au-dessus du maximum possible du terrain */
→       if(cur.z > (param.hautmax + L_DIAM_RAYON))
→           {
→           /* On sort, pas d'intersection possible */
→           break;
→           }

→       /* Si le terrain est au-dessus du vecteur de lumière, alors le terrain
→       cache la lumière */
→       if(tmp > cur.z + L_DIAM_RAYON)
→           {
→           return 0.0;
→           }

→           /*Calcul du facteur de visibilité de soleil*/
→         newVsun = ((cur.z + L_DIAM_RAYON) - tmp)/(2*L_DIAM_RAYON)
→           if(newVsun < Vsun)
→               {Vsun = newVsun;
→               }

→       /* Sinon, on continue avec le vecteur de lumière */
```

$\rightarrow$ $cur.x+ = light- > x;$

$\rightarrow$ $cur.y+ = light- > y;$

$\rightarrow$ $cur.z+ = light- > z;$

$\rightarrow$ $\}$

$\rightarrow$ $/* \ Sinon\ pas\ d'intersection\ ,\ on\ calcul\ la\ luminosité\ avec\ le\ calcul\ L\ dot\ N\ */$

$\rightarrow$ $return\ Vsun * Terrain\_calcLightMap\_LDOTN(image, i, j, maxi, maxj);$

$\rightarrow\}$

La fonction $Terrain\_calcLightMap\_LDOTN$ est bien présentée dans la section 3.5.3.

L_DIAM_RAYON n'est pas le diamètre perpendiculaire $(R = 2r)$ mais il représente le r (demi diamètre perpendiculaire).

Le résultat illustrant cette solution est donné par l'image suivante :

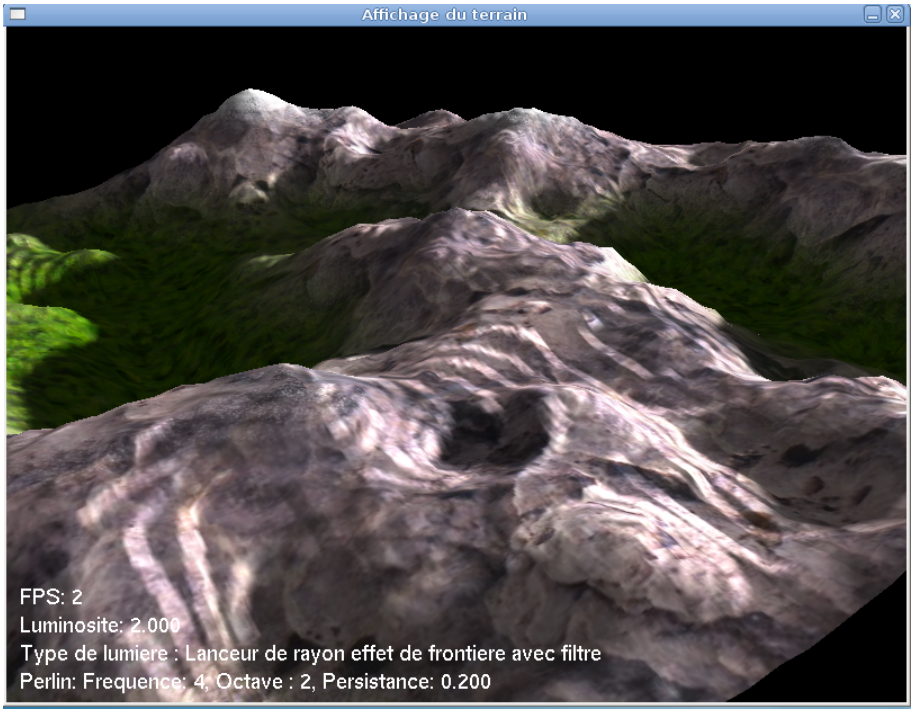


FIG. 4.4 – Résultat de la première version

Mais nous remarquons que ce résultat est totalement incohérent, cela implique qu'il ya un problème.

Voir ce schéma :

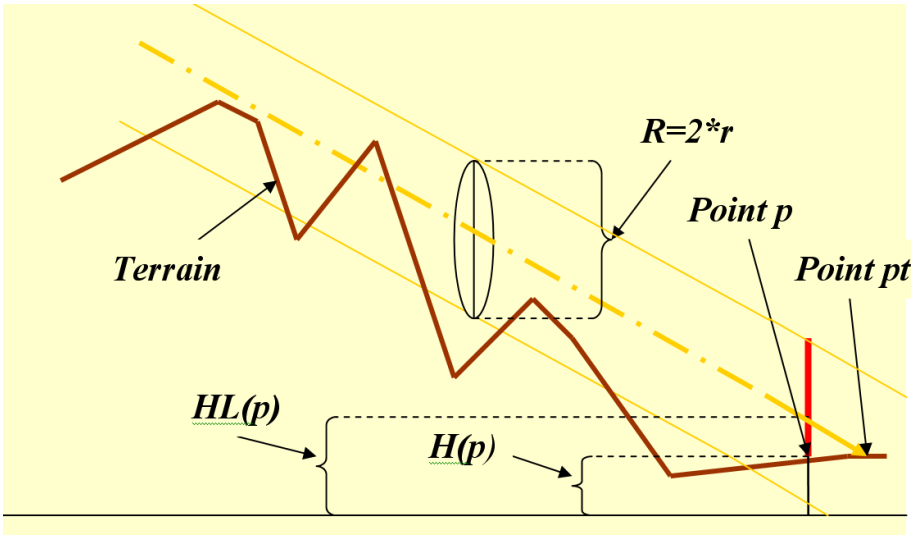


FIG. 4.5 – Problème de la première solution

Nous remarquons bien que dès le départ une partie de la surface du vecteur lumière est cachée par les voisins du point de travail pt, ce qui rend le soleil partiellement visible dans toutes les situations, produisant de l'ombre sur tout le terrain.

4.4.2 Version finale de l'approche proposée

Comme nous avons vu, si nous considérons que le vecteur lumière est un cylindre, nous aurons le problème que les voisins du point de travail cachent partiellement le vecteur lumière, dans cette section nous allons proposer une solution résolvant ce problème.

Au lieu de considérer que le vecteur lumière est un cylindre, nous allons supposer que c'est un cône avec un diamètre de longueur variable.

La longueur de diamètre commence de la valeur 0, la valeur augmente linéairement avec le déplacement du vecteur lumière ; à chaque pas ΔX nous ajoutons une valeur Δr au diamètre du vecteur lumière.

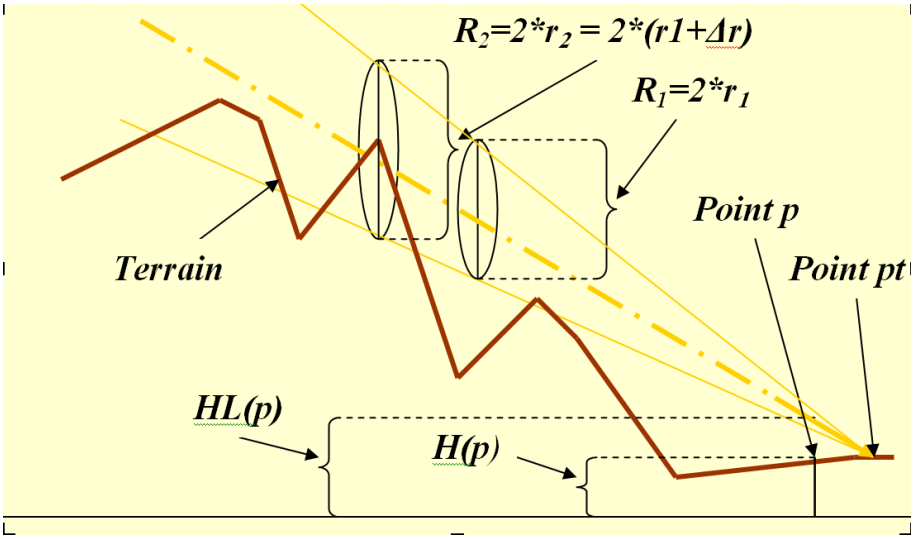


FIG. 4.6 – Principe de cône

Le pseudo-code de la fonction qui réalise cette proposition est le suivant :

$\rightarrow$ *double Terrain_calcLightMap_RayFront(SDL_Surface *image,int i, int j, int maxi, int maxj, SVecteur *light)*

$\rightarrow$ {

$\rightarrow$ *SPoint cur;*

$\rightarrow$ *float tmp;*

$\rightarrow$ *double Vsun,newVsun;*

$\rightarrow$ */**

$\rightarrow$ **Verification s'il y a une intersection avec le terrain, si*

$\rightarrow$ ** c'est le cas, il n'y aura pas de lumiere ici*

$\rightarrow$ **/*

$\rightarrow$ */**

$\rightarrow$ ** Point de depart, le point(i,j) avec son hauteur*

$\rightarrow$ **/*

$\rightarrow$ *tmp = i;*

$\rightarrow$ *tmp/ = maxi;*

$\rightarrow$ *tmp* = image− > h;*

```
→       cur.x = tmp;

→       tmp = j;
→       tmp/ = maxj;
→       tmp* = image- > w;

→       cur.y = tmp;
→       cur.z = Terrain_GetHauteur(image, i, j, maxi, maxj);
→       Vsun = 1;
→       newVsun = 2;
→       double L_DIAM_RAYON = 0.0;

→       / * Tant qu'on est dans le terrain * /
→       while((cur.x >= 0)&&(cur.y >= 0)&&(cur.x < image- > h)&&
→           (cur.y < image- > w))
→           {
→           / * On recupère la hauteur courante * /
→           tmp = Terrain_GetHauteur(image, (int)cur.x, (int)cur.y, image- >
→           h, image- > w);

→           / * Si c'est au - dessus du maximum possible du terrain * /
→           if(cur.z > (param.hautmax + L_DIAM_RAYON))
→               {
→               / * On sort, pas d'intersection possible * /
→               break;
→               }

→         / * Si le terrain est au - dessus du vecteur de lumière, le terrain cache la lumière *
/
→           if(tmp > cur.z + L_DIAM_RAYON)
→               {
→               return 0.0;
```

```
→                }

→                /* Calcul du facteur de visibilité de soleil */
→               if(L_DIAM_RAYON!=0.0)newVsun=((cur.z+L_DIAM_RAYON)
→                    -tmp)/(2*L_DIAM_RAYON);
→                if(newVsun<Vsun)
→                    {Vsun=newVsun;
→                    }
→                L_DIAM_RAYON+=param.Delta_r; /* Ajout de Δr au diamètre */

→            /* Sinon, on continue avec le vecteur de lumière */
→            cur.x+=light->x;
→            cur.y+=light->y;
→            cur.z+=light->z;
→           }

→       /* Sinon pas d'intersection , on calcul la luminosité avec le calcul L dot N */
→       return Vsun*Terrain_calcLightMap_LDOTN(image,i,j,maxi,maxj);
→}
```

Nous remarquons que la valeur de diamètre commence du 0, et à chaque pas (itération) on lui ajoute la constante $param.Delta\_r$ qui représente le Δr.

Voici une image illustrant cette solution :

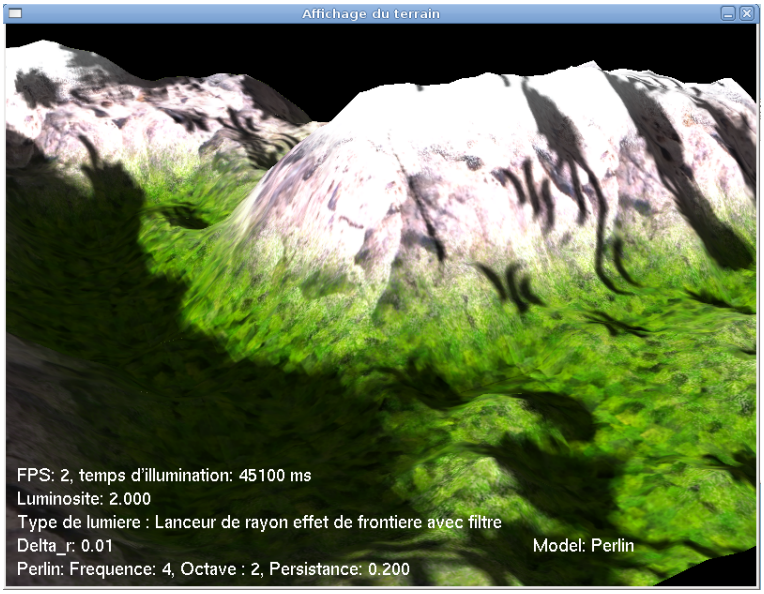


FIG. 4.7 – Solution de la version finale

Si nous comparons cette image avec la solution de la méthode de lanceur de rayon pour le calcul des ombres présentée dans la section 3.5.4 :

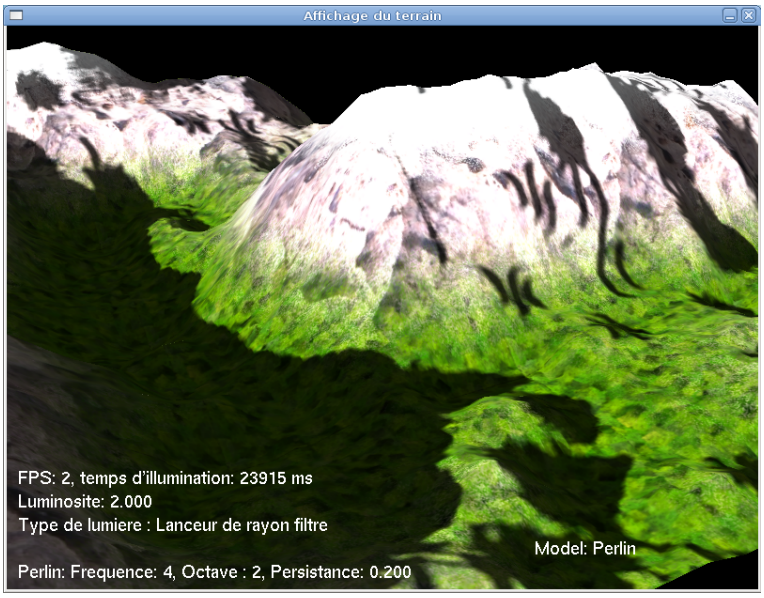


FIG. 4.8 – solution de la méthode de lanceur de rayon

nous remarquons que la différence est bien visible dans les frontières des ombres, le terrain généré par la méthode proposée possède des frontières d'ombres claires.

4.5 Conclusion

Dans ce chapitre nous avons essayé d'introduire une nouvelle approche d'illumination de terrain, nous avons essayé d'accroitre le réalisme sans augmenter la complexité d'une manière considérable, ce qui a engendré des calculs supplémentaires uniquement pour le facteur de visibilité, et sans avoir besoin de structures supplémentaires et complexes tel que la carte horizon.

Pour la modélisation nous allons évaluer et discuter dans le chapitre suivant les résultats obtenus par la technique mixte proposée. Pour les deux techniques améliorées, elles donnent des résultats excellents surtout la dernière qui possède le réalisme de Perlin et la vitesse de génération.

Chapitre 5

Résultats et perspectives

5.1 introduction

Dans ce chapitre nous allons présenter un ensemble de résultats permettant de valider les différentes méthodes propsées dans ce travail, ces résultats sont obtenus en utilisant un ordinateur qui possède les caractéristiques suivantes :

avec une carte graphique intégrée intel(R) 32 Mo.

nous avons réutilisé le code de viewer de terrain de Fearyourself téléchargeable à partir du lien http://ftp-developpez.com/khayyam/articles/algo/perlin/terrain_viewer.zip , afin d'implémenter nos modules et de les intéger dans ce viewer.

5.2 Génération de terrain

Pour les résultats obtenus nous avons utilisé l'algoritme de Perlin avec les paramètres suivant : fréquence : 4, octaves : 2 et une persistance de 0.2.

Et voici une image qui montre un terrain affiché en mode filaire :

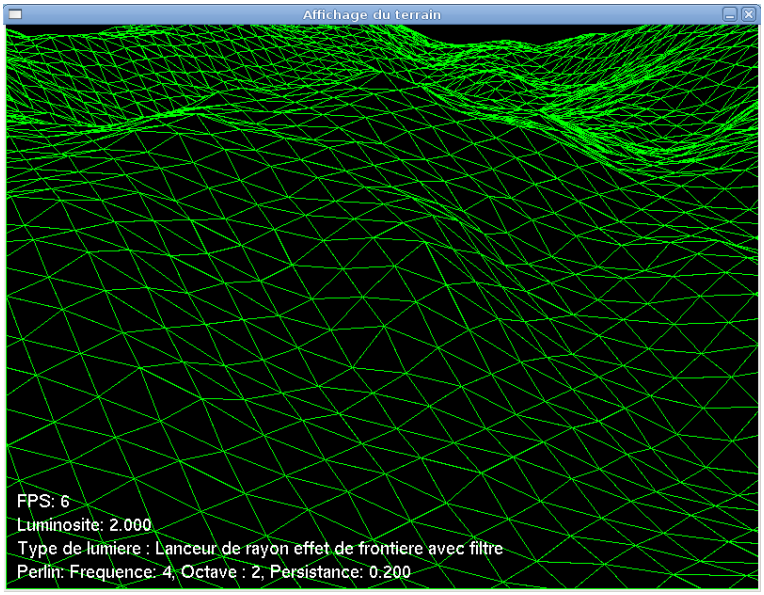


FIG. 5.1 – Terrain de perlin en mode filaire

5.3 Placage de texture

La texture plaquée sur le terrain modélisé est générée par la méthode de mélange de textures de base et voici un terrain aprés le placage de texture :

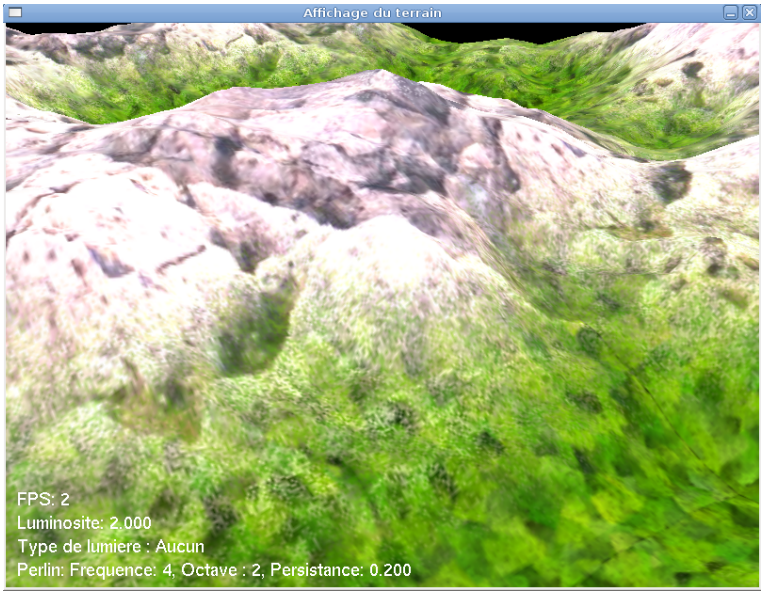
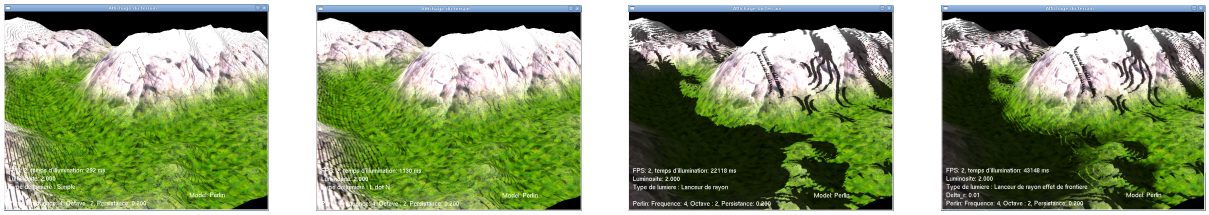


FIG. 5.2 – Terrain de perlin texturée

5.4 Illumination

pour le calcul du rendu nous avons utilisés une luminosité de 2.0 avec un terrain de Perlin ayant les paramètres : $Fréquence = 4$, $Octave = 2$, $Persistance = 0.2$, pour toutes les méthodes l'utilisation de filtre est nécessaire, pour éliminer les imperfections.

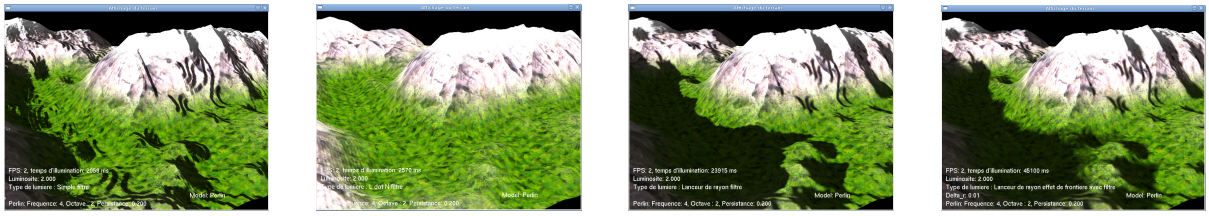
Les résultats obtenus sans l'utilisation du filtre :



(1) (2) (3) (4)

FIG. 5.3 – Résultats de l'illumination sans utiliser le filtre

Et voici les résultats aprés l'utilisation du filtre :



(1) (2) (3) (4)

FIG. 5.4 – Résultats de l'illumination avec l'utilisation du filtre

(1) : Première version (Simple)

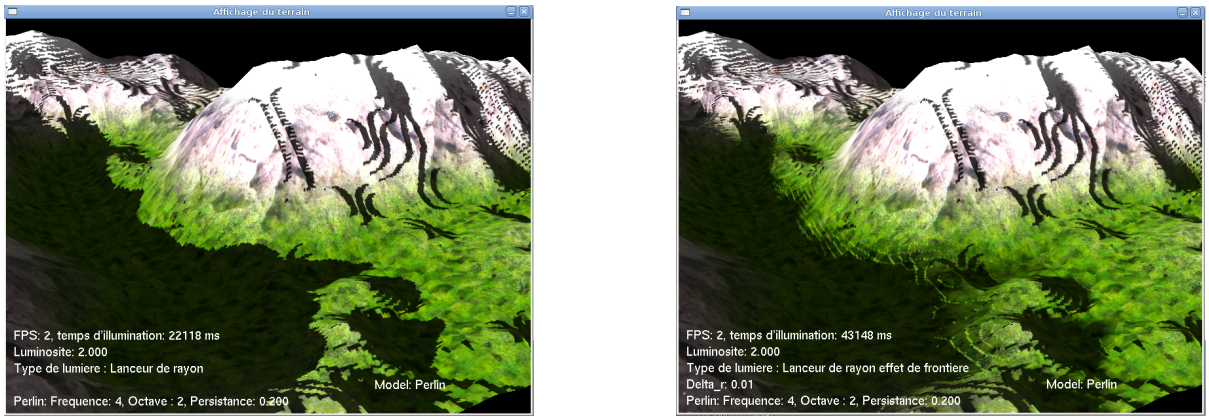
(2) : Deuxième solution (L dot N)

(3) : Dernière méthode (Lanceur de rayon)

(4) : version finale proposée (Lanceur de rayon effet de frontière)

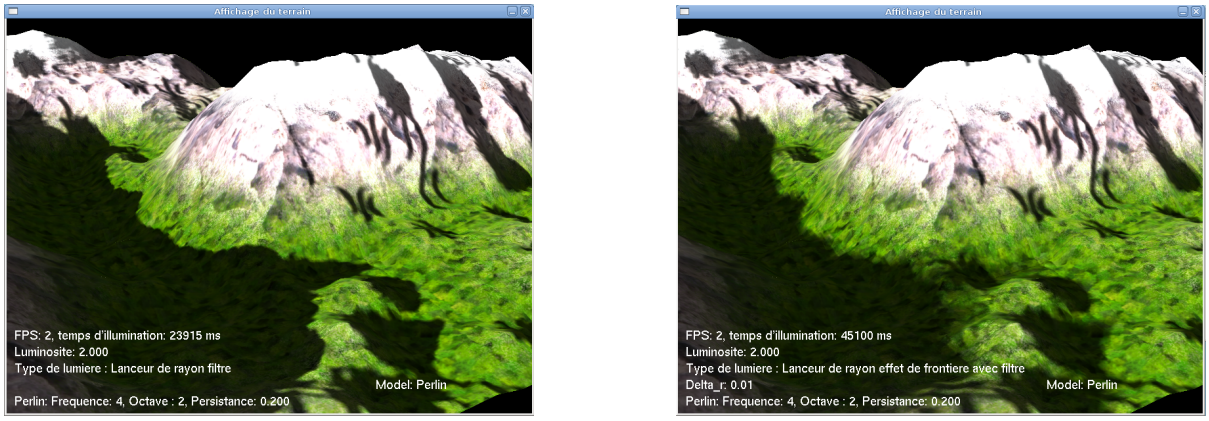
5.4.1 Résultats

Nous allons comparer notre méthode avec la version la plus réccente (Lanceur de rayon)



(1) (2)

FIG. 5.5 – Illumination sans utiliser le filtre



(1) (2)

FIG. 5.6 – illumination avec l'utilisation du filtre

(1) : Dernière méthode (Lanceur de rayon)

(2) : version finale proposée (Lanceur de rayon avec effet de frontière)

On remarque bien que la méthode proposée (Lanceur de rayon avec effet de frontière) possède un effet supplimentaire ; les frontières d'ombres sont claires, mais elle est relativement lente.

5.4.2 Modélisation de lumière de la nuit

Généralement les sources lumineuses de la nuit tel que la lune les lampes sont plus proches que le soleil, alors leur diamètre angulaire est plus grand, et elles ont une intensité lumineuse faible, donc pour modéliser la lumière de la nuit, il suffit de mettre le paramètre Δr plus grand (ici $\Delta r = 1.5$) dans la méthode proposée, et réduire la luminosité à 1.
Et voici une image qui illustre la lumière de la nuit :

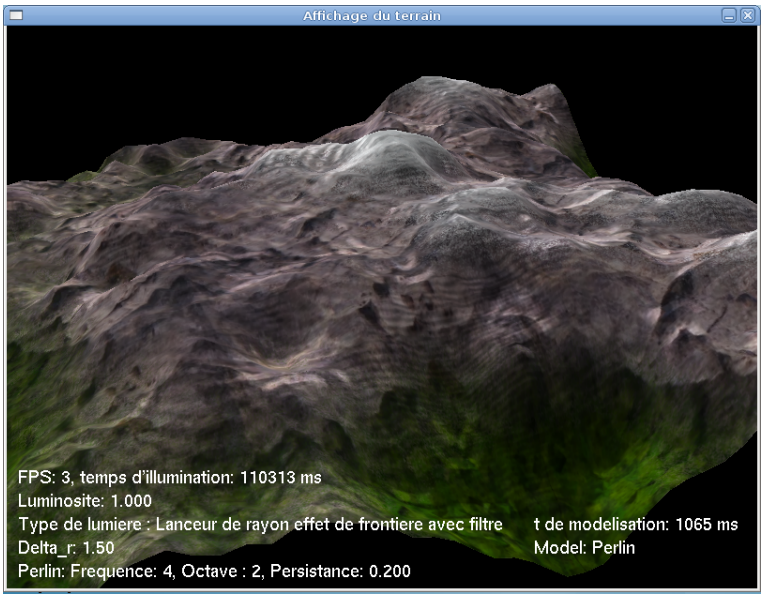


FIG. 5.7 – Lumière de nuit

5.5 Modélisation

Pour toutes les figures que nous allons utiliser pour montrer les résultats de modélisation, nous avons utilisés pour le rendu la nouvelle méthode de calcul d'ombre proposée, avec $\Delta r = 0.01$.
Comme nous avons montré dans le chapitre précédent les petites crêtes sont réduites par la technique de modélisation mixte que nous avons proposée, comme elles montrent les figures suivantes :

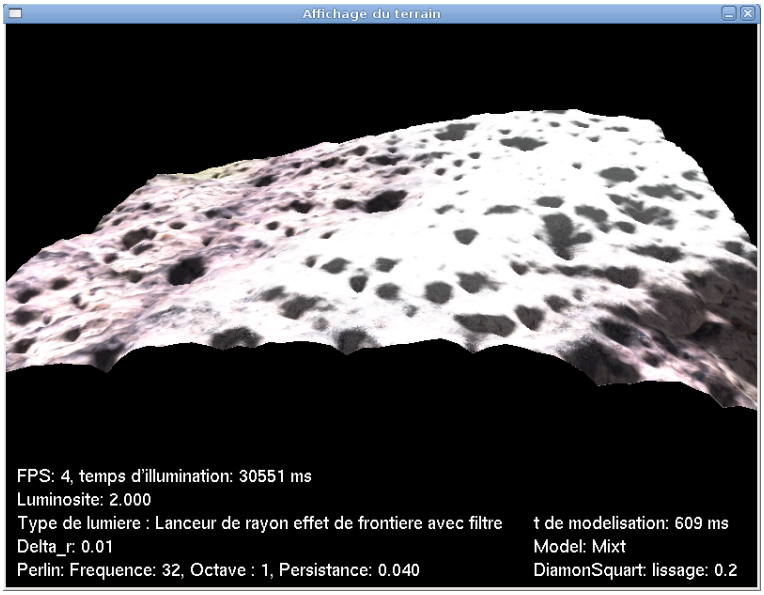


FIG. 5.8 – Modèle mixte

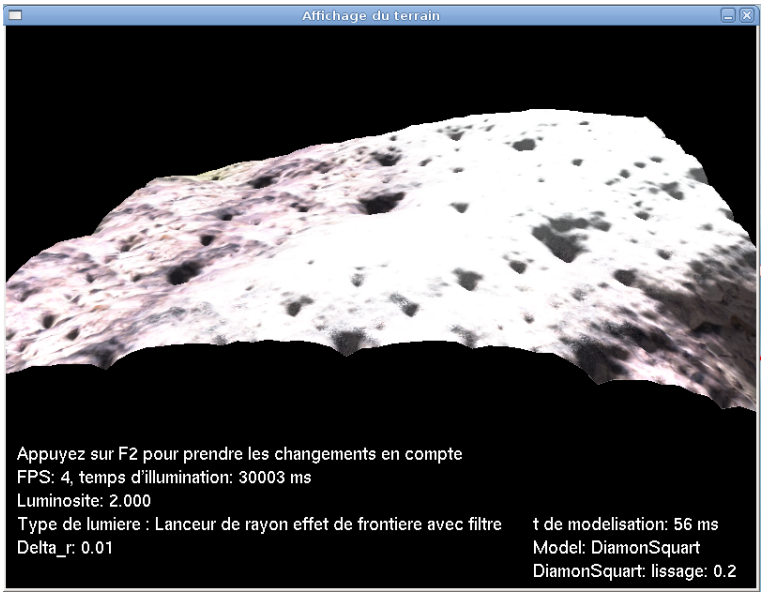


FIG. 5.9 – Modèle Diamond-Square

Avec cette texture, il se peut qu'il y est confusion entre les détails de la texture et le relief de terrain, nous allons donc essayer d'ululiser une texture moins détaillée.

Nous comparons les résultats maintenant :

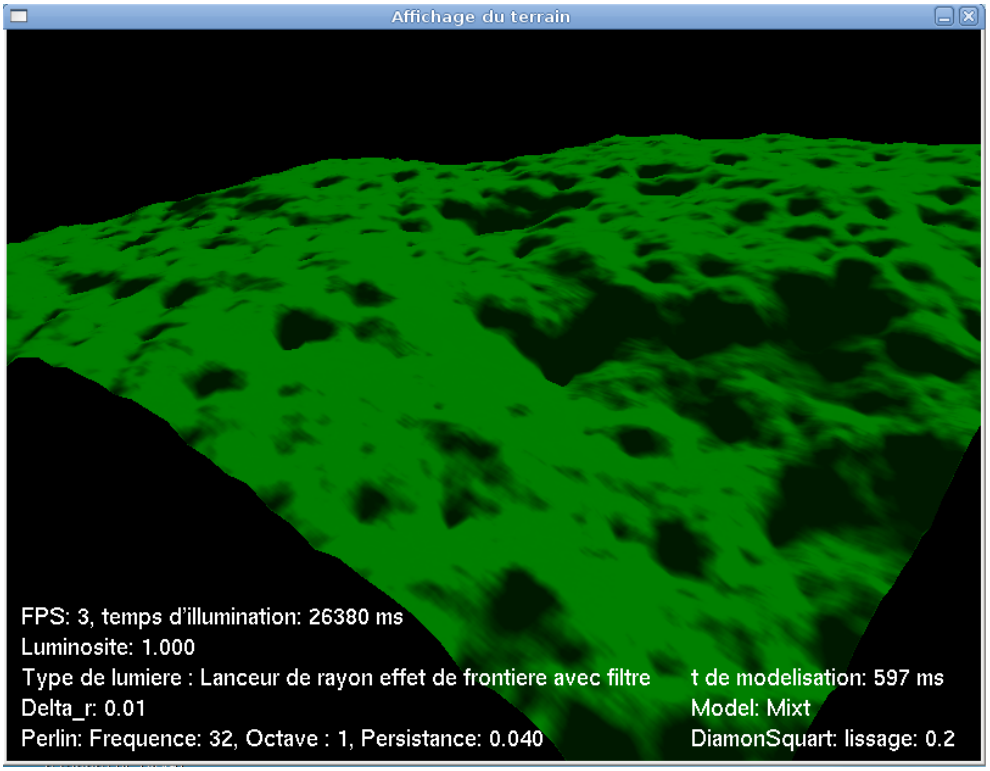


FIG. 5.10 – Modèle mixte à texture de faible détail

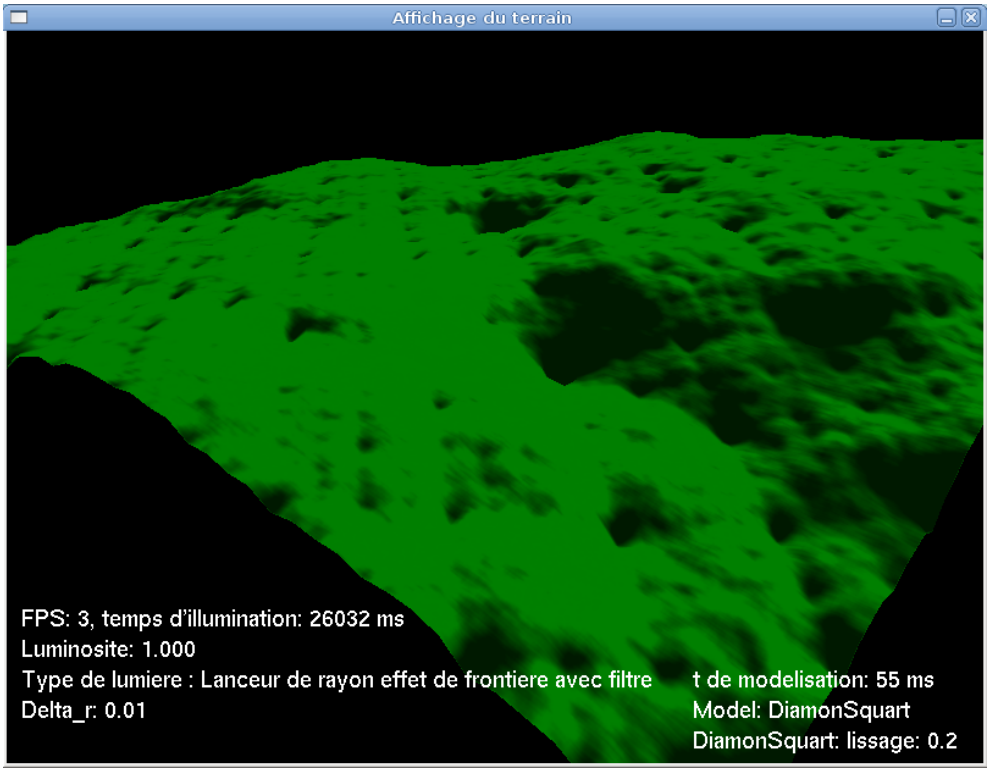
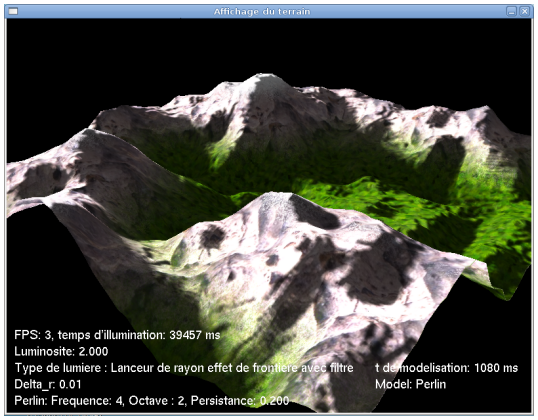
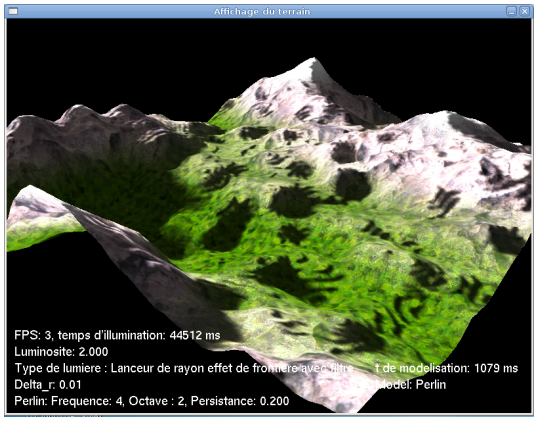


FIG. 5.11 – Modèle Diamond-Square à texture de faible détail

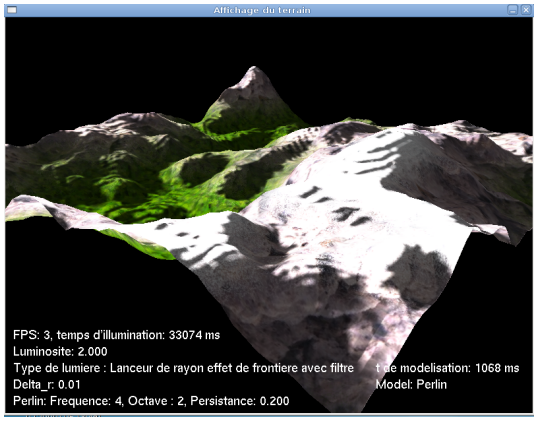
Maintenant, il ne reste plus qu'a comparer les vitesses des méthodes, et voici des terrains générés par les différentes métodes possèdent le même nombre de points :



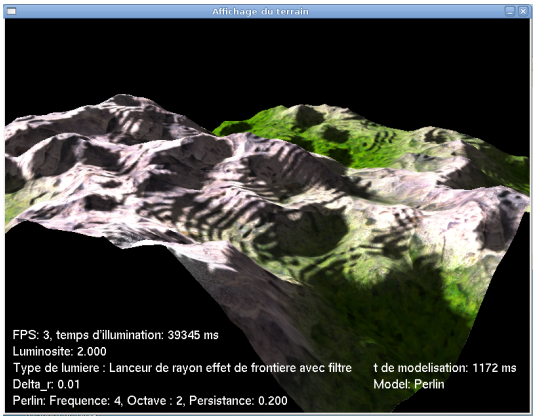
(1)



(2)



(3)

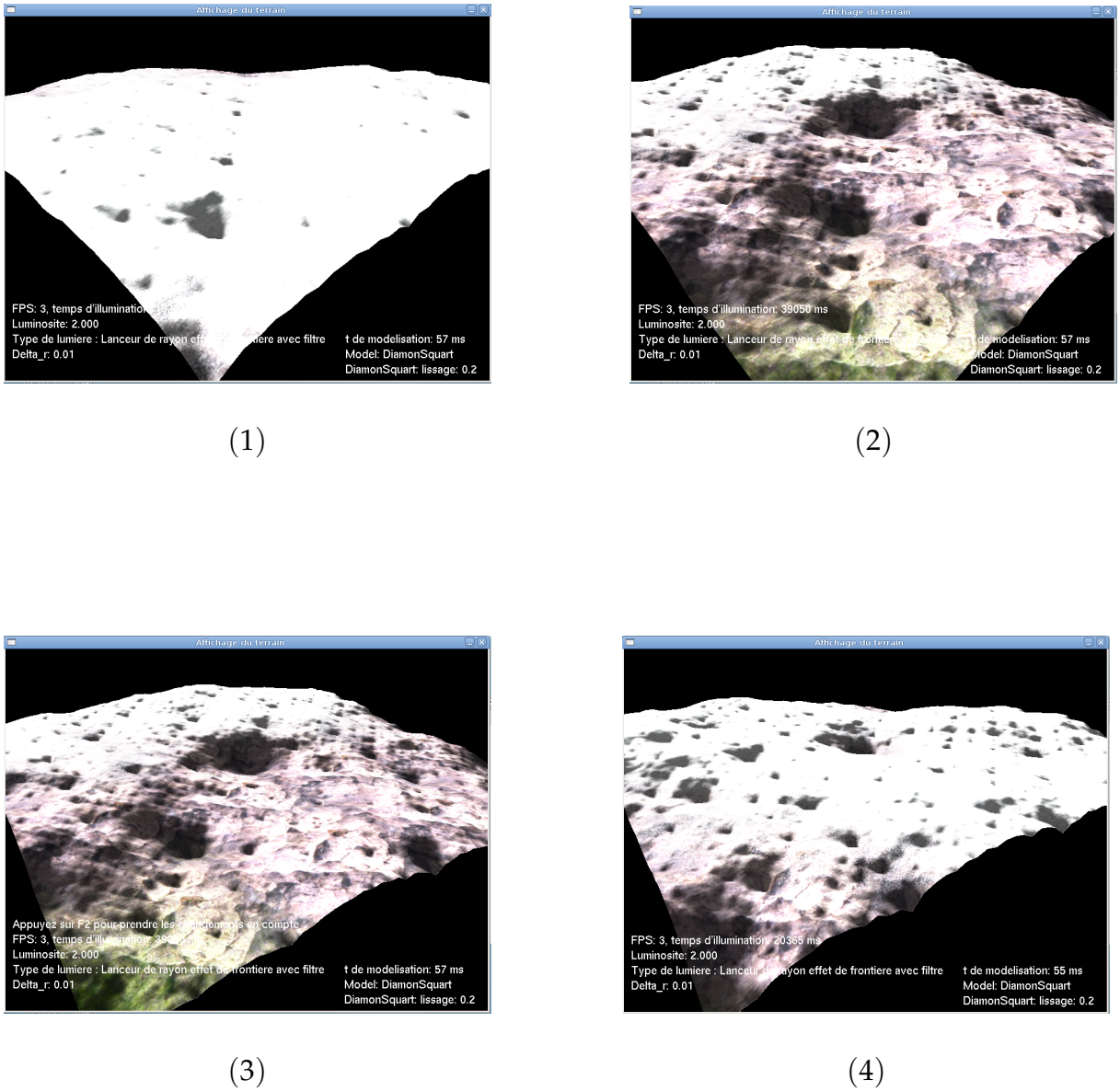


(4)

FIG. 5.12 – Résultats de Perlin

Paramètres

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



(1) (2)

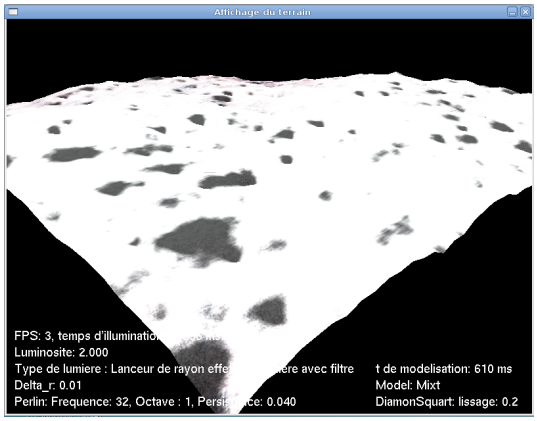
(3) (4)

FIG. 5.13 – Résultats de Diamond-Square

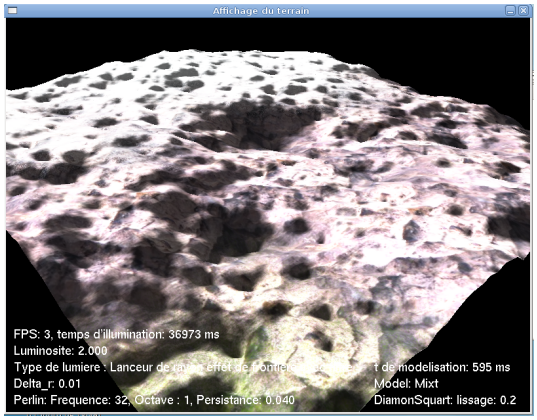
Paramètres

– Le coefficient de lissage est 0.2.

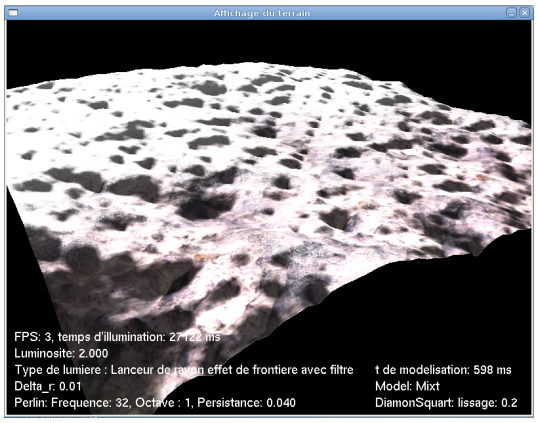
Nous avons choisis 0.2 parce que les petites crêtes sont visible avec un terrain lisse.



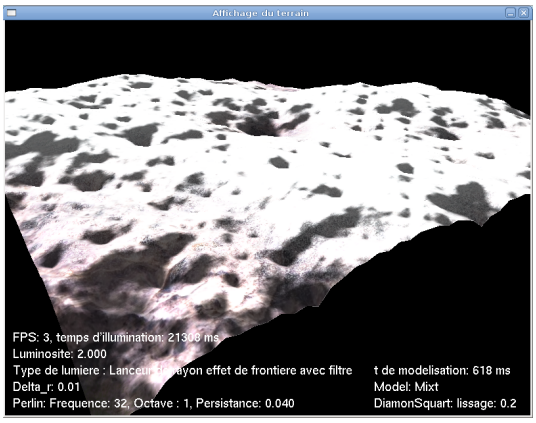
(1)



(2)



(3)



(4)

FIG. 5.14 – Résultats du modèle mixte

Paramètres du composante de détails générée par Perlin

- La fréquence de départ est 32 ;
- Le nombre d'octaves est 1 ;
- La persistance de départ est 0.04.

Paramètres du composante globale générée par Diamond-Square

- Le coefficient de lissage est 0.2.

Voici un tableau qu illustre les temps de modélisation des différents terrains :

Méthode	Terrain (1)	Terrain (2)	Terrain (3)	Terrain (4)	Moyenne
Perlin	1080ms	1079ms	1068ms	1172ms	1099ms
Diamond-Square	57ms	57ms	57ms	55ms	56.5ms
Mixte	610ms	595ms	598ms	618ms	603ms

D'après les résultats il est claire que la méthode proposée plus rapide que la méthode de perlin.

5.6 Exemple d'exploitation des résultats (simulateur de vol)

Après l'obtention des résultats, nous allons exploiter ces résultat, afin de les mettre dans leurs cadre pratique.

Supposant que nous avons un large terrain utilsé comme paysage dans un simulateur de vol, un utilisateur qui a l'habitude d'utiliser ce simulateur va certainement apprendre les détails, alors son comportement ne sera pas réactif, ceci rend l'utilisation de la méthode basée sur la carte horizon non efficace, car elle se base sur un model et un carte horizon pré-établis, aussi l'illumination d'un vaste terrain necéssite une phase d'initialisation de simulateur.

La solution et d'utiliser une technique avec laquelle on peut modéliser et faire le rendu d'une manière intéractive, cette solution consiste à suivre les étapes suivantes :

- Génération de la forme globale du terrain par un calque de Perlin à faible fréquence.
- Division du terrain en sous parties plus petites.
- Génération d'une seul composante détaillé par des couche de calque de Perlin
- Quand l'utilisateur arrive à une sous partie, il faut appliquer la composante de détails au model de la partie en question et les partie voisins, et faire le rendu seulement de la partie en question.

Ce schéma ilustre ces étapes :

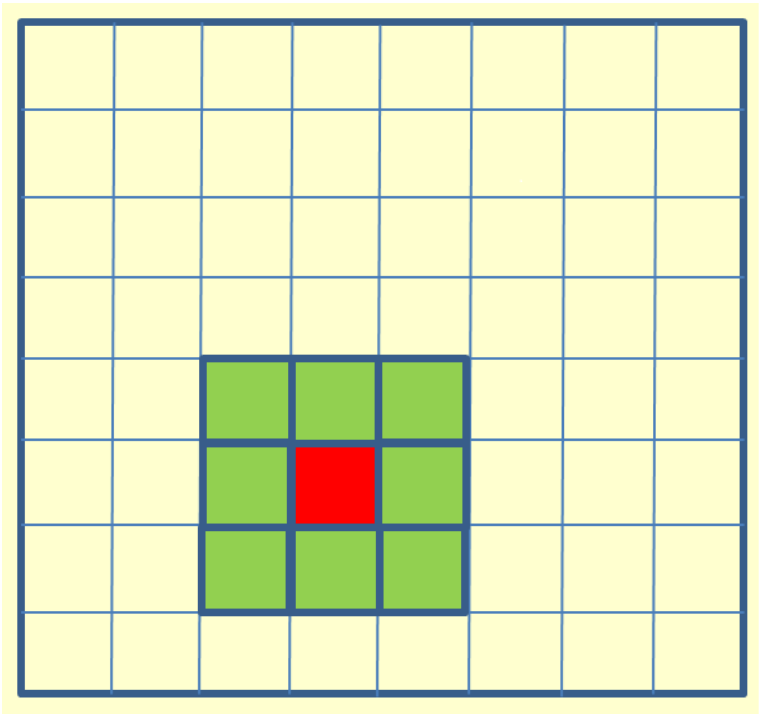


FIG. 5.15 – Principe de simulateur de vol

Nous avons appliqué la composante de détails au models des parties voisins, car leurs forms affectent le rendu de la partie en question, pour l'illumination nous proposons d'utilser la technique de calcul d'ombre que nous avons proposé

5.7 Perspectives

Les pricipales améliorations des techniques que nous avons présenté, consistent essentiellement à pallier au problème lié ou coût de calcul, tout en gardant une qualité visuelle acceptable pour la modélisation et le rendudes terrains ; nous envisageons un ensemble d'améliorations et d'extensions pour nos travaux, qui peuvent être :

- Pour éliminer les imperfections nous avons utilisé un filtre, alors la solution obtenue n'est pas la vraie solution, nous pouvons réduire les imperfection en augmentant la densité des points par l'accroissement des dimentions de heightmap.
- Nous remarquons que les ombres dans les méthodes locales ont une couleur noir mate parfaite, ceci est dûe à l'élimination des inter-réflexions entre les points du terrain par ces méthodes, nous pouvons approximer l'effet des inter-réflexions sans avoir besion d'un modèle globale.

– L'accélérations via les cartes graphiques par l'utilisation des shaders.

5.8 Conclusion

Ce chapitre est consacré a la présentation des résultats permettant d'évaluer les perfermences des nouvelles méthodes proposées, nous avons vu que chaque méthode a un défaut et chaque méthode vient pour résoudre le défaut de la précédente.

Pour l'illumination, la nouvelle méthode de calcul d'ombre proposée est plus réaliste que les méthodes présentées dans l'état de l'art, mais relativement lente.

Pour la modélisation la méthode mixte proposée est plus rapide que la méthode de Perlin, elle a aussi réduit et presque éliminé le problème des crêtes posé par la méthode Diamond-Square.

Conclusion générale

A travers ce travail, nous avons présenté le rendu des terains, Les terrains jouent un rôle fondamental dans la création d'une scène naturelle. Les techniques de génération automatique ont de nombreuses applications comme les simulateurs de vol, les effets spéciaux au cinéma ou les jeux vidéo. L'augmentation de la puissance de calcul combinée à des méthodes de visualisation de plus en plus performantes ont permis de créer des terrains réalistes.

En réalisant ce travail, nous avons pu acquérir :

- Des connaissances sur la modélisation des terrains qui ouvre une grande porte vers la recherche des techniques spéciales pour générer les terrains.
- Des idées générales sur la génération de textures.
- Quelques techniques d'illumination de terrain.

Nous avons exploité ces connaissances pour développer de nouvelles méthodes plus réalistes, et plus intéractives, nous avons **implémenté une nouvelle méthode pour la modélisation et une autre pour l'illumination**. Les résultats d'évaluation sont satisfaisants.

L'objectif global du projet est atteint. Cependant, plusieurs **perspectives** commencent à être envisagées. Nous pensons que ce travail peut être une brique de base pour des travaux plus affinés qui peuvent traiter :

- Simuler plus d'effets de réalisme tel que les inter réflexions par approximation sans utiliser des modèls globaux et augmenter la complexité.
- Intégrer l'aspect multirésolution (LOD, programmation via le matériel graphique, ...).
- Possibilité d'intégrer des objets (conçue par un logiciel graphique) telque les arbres dans le terrain de synthèse .

Les fondements théoriques pour ces différents axes sont élaborés. Mais un travail important est exigé, pour passer de cet aspect théorique, vers une réalisation fructueuse.

Bibliographie

[STE96] Stephen Richard Cook, Polygonal approximation of terrain across load module boundaries with constrained Delaunay Triangulation, 1996.

[AND03] André Lamothe Focus on 3D Terrain Programming, Premier Press, 2003.

[BEN77] Benoit Mandelbrot, Fractals form chance and dimension, Freeman,1977.

[LAU93] Laura Lang, Terrain Modeling, Computer Graphics World, 16(9), p. 22- ? ?, September 1993.

[FB91] F. Musgrave and B. Mandelbrot, The art of fractal landscapes, IBM Journal of Research and Development, 35(4), pp. 535-540, July 1991.

[GJE84] G. J. Edwards, Fractal Based Terrain Modelling, Computer FX '84. Proc. the Conference on Computer Animation and Digital Effects, pp. 49-56, 1984.

[KEN02] Ken Perlin, Better acting in computer games : the use of procedural methods, Computers and Graphics, 26(1), pp. 3-11, February 2002.

[CLI95] Clifford A. Reiter,Synthetic coloring of fractal terrains and triangular automata, The Visual Computer, 11(6), pp. 313-318, Springer-Verlag, 1995.

[NK01] Naty Hoffman, Kenny Mitchell, Real-Time Photorealistic Terrain Lighting, Westwood Studios, 2001.

[DAN03] Daniel Thalmann, Infographie, Ecole Polytechnique Fédérale de Lausanne, 2003.

[AEJS08] A. Peytavie, E. Galin, J. Grosjean, S. Merillou , Modélisation de terrains complexes 3D, Toulouse, 2008.

[MGD00] Manuel M. Oliveira and Gary Bishop and David McAllister, Relief Texture Mapping, Siggraph 2000, Computer Graphics Proceedings, Annual Conference Series, pp. 359-368, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[YX09] Y. Amara and X. Marsault, A GPU Tile-Load-Map architecture for terrain rendering : theory and applications, The Visual Computer, 25(8), p. xx-yy, August 2009.

[MAN82] MANDELBROT B, The Fractal Geometry of Nature, W. H. Freeman, August 1982.

[MKM89] MUSGRAVE F. K., KOLB C. E., MACE R. S. : The synthesis and rendering of eroded fractal terrains. In Proceedings of ACM SIGGRAPH (1989), pp. 41-50.

[ST89] SZELISKI R., TERZOPOULOS D. : From splines to fractals. In Proceedings of ACM SIGGRAPH (1989), pp. 51-60.

[JS06] JENS SCHNEIDER T. BOLDTE R. W. : Real-time editing, synthesis, and rendering of in ?nite landscapes on gpus. In Conference on Vision, Modeling, and Visualization (2006), pp. 153- 160.

[EMP*98] E BERT D., MUSGRAVE K., PEACHEY D., PERLINK., WORLEY S. : Texturing and Modeling : A Procedural Approach. Academic Press Professional, 1998.

[GM01] G AMITO M., MUSGRAVE F. K. : Procedural landscapes with overhangs. In 10^{th} Portuguese Computer Graphics Meeting, held in Lisbon (2001).

[NWD05] N EIDHOLD B., WACKER M., DEUSSEN O. : Interactive physically based ?uid and erosion simulation. In Eurographics Workshop on Natural Phenomena (2005), pp. 25-32.

[KMN88] KELLEY A. D., MALIN M. C., NIELSON G. M. : Terrain simulation using a model of stream erosion. In Proceedings of ACM SIGGRAPH (1988), pp. 263-268.

[STE98] A.J. Stewart. Fast horizon computation at all points of a terrain with visibility and shading applications. IEEE Transactions on Visualization and Computer Graphics, 4(1) :82–93, March 1998.

[RPP93] R OUDIER P., PEROCHE B., PERRIN M. : Landscapes synthesis achieved through erosion and deposition process simulation. Computer Graphics Forum 12, 3 (1993), 375-383.

[CMF98] CHIBA N., MURAOKA K., FUJITA K. : An erosion model based on velocity ?elds for the visual simulation of mountain scenery. Journal of Visualization and Computer Animation 9, 4 (1998), 185-194.

[MDH07] MEI X., DECAUDIN P., HU B. : Fast hydraulic erosion simulation and visualization on gpu. In Pacific Graphics (2007), pp. 47-56.

[IFMC03] ITO T., FUJIMOTO T., MURAOKA K., CHIBA N. : Modeling rocky scenery taking into account joints. cgi 00 (2003), 244.

[BFO07] B EARDALL M., FARLEY M., OUDERKIRK D., SMITH J., JONES M., EGBERT P. : Goblins by SpheroidalWeathering. In Eurographics Workshop on Natural Phenomena (2007), pp. 7-14.

[ZSTR07] Z HOU H., SUN J., TURK G., REHG J. M. : Terrain synthesis from digital elevation models. IEEE Transactions on Visualization and Computer Graphics 13, 4 (July/August 2007), 834-848.

[MAX88] N.L. Max. Horizon Mapping : Shadows for Bump-Mapped Surfaces. The Visual Computer, 4(2) :109-117, July 1988.

[AM97] A.J. Stewart., M.S. Langer. Towards Accurate Recovery of Shape from Shading under Diffuse Lighting. IEEE Transactions on Pattern Analysis and Machine Intelligence, 19(9) :1020-1025, Sept. 1997.

[WKJH00] W. Heidrich, K. Daubert, J. Kautz, and H.-P. Seidel. Illuminating micro geometry based on precomputed visibility. Computer Graphics(Proceedings of SIGGRAPH 2000) :455-464, July 2000.

[PM00] P.-P. Sloan, M.F. Cohen. Interactive Horizon Mapping. Rendering Techniques 2000(Proceedings of Eurographics Rendering Workshop 2000) : 281-298, June 2000.

[MIT] K. Mitchell. Real-Time Full Scene Anti-Aliasing for PCs and Consoles. Proceedings of Game.

[STE98] A. James Stewart, Fast Horizon Computation at All Points of a Terrain With Visibility and Shading Applications, IEEE Transactions on Visualization and Computer Graphics, 4(1), p. 82- ?, January 1998.

[JOH07] Johan Andersson, Terrain Rendering in Frostbite Using Procedural Shader Splatting, SIGGRAPH, 2007.

[MA98] Marc Olano and Anselmo Lastra, A Shading Language on Graphics Hardware : The PixelFlow Shading System, SIGGRAPH 98 Conference Proceedings, Annual Conference Series, pp. 159-168, Addison Wesley, July 1998.