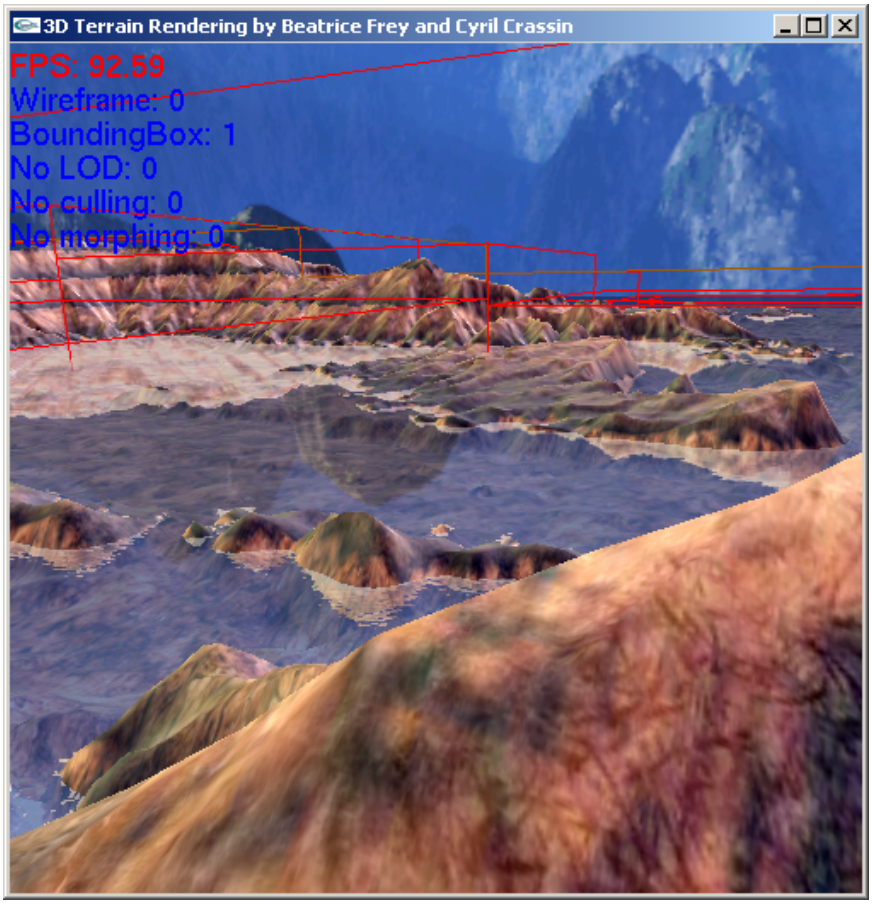
Projet IN55

Visualisation de terrain



Sommaire

Introduction

L'objectif global de ce projet était de mettre en œuvre une visualisation de terrains définis à partir de cartes de hauteur (*height map*). Il s'agit en effet du moyen le plus courant pour définir un terrain géographique. Le but était de pouvoir réaliser cette visualisation en temps réel sur de vastes étendues de terrain extrêmement détaillées.

Cette visualisation passe par la génération d'un maillage capable d'être traité et visualisé au travers des pipelines graphiques standards proposés par les API modernes de visualisation telles que OpenGL qui a été utilisé dans ce projet.

Afin de permettre une visualisation interactive de ces grands espaces complexes, diverses techniques de partitionnement de l'espace (sélection des zones visibles) et de simplification ont dues être utilisées.

Une attention particulière a été portée au choix et à la mise en œuvre de l'algorithme de simplification utilisé. Le projet a donc débuté par la réalisation d'un état de l'art sommaire mais qui nous a permis de choisir la technique la mieux adaptée aux contraintes que nous nous étions fixées. Nous voulions en effet pouvoir tirer parti au mieux des capacités du matériel graphique à notre disposition. Le but était également de pouvoir mettre en œuvre un algorithme suffisamment souple pour être utilisé dans un contexte de très gros volumes de données ne tenant pas entièrement en mémoire.

Dans le but d'augmenter le réalisme du rendu, nous nous sommes également intéressés au mapping et à l'application d'un éclairage dynamique sur ce terrain. Un système de matériel paramétrique, basé sur un jeu de textures représentant les éléments naturels constituant le terrain, a été mis en œuvre.

Au niveau de l'éclairage, une illumination par fragment a été implémentée. Ce type d'illumination permet d'ajouter encore un niveau de réalisme au rendu en simulant un relief sur des surfaces peu polygonalisées. Afin de pouvoir gérer encore une fois de très vastes étendues de terrain, la simplification des textures utilisées pour cet éclairage a été mise en œuvre sur le même modèle que celle utilisée pour le maillage.

1. Techniques de simplification

Il existe deux grandes familles de techniques de niveaux de détails. Les niveaux de détails « simples » ou LOD, utilisés principalement pour l'affichage de petits objets, permettent de définir un ensemble de maillages plus ou moins détaillés parmi lesquels on choisit dynamiquement le mieux adapté en fonction de la distance du modèle au point de vue.

Ce type de technique n'est pas adapté au rendu de terrains qui sont modélisés par un unique maillage constamment présent à l'écran et qu'il est donc impossible de simplifier globalement. Il faut en effet plus ou moins détailler les zones en fonction de leur proximité au point de vue et également leur relief.

Pour cela, des techniques dites à niveaux de détails « continus » (*CLOD, Continuous Level Of Detail*) ont été étudiées et développées. Ces techniques permettent de choisir un niveau de détail en tout point du maillage en fonction de divers critères, il s'agit généralement de la distance au point de vue et d'un critère permettant d'évaluer la niveau d'accident du relief (un relief plus accidenté nécessitant plus de détails).

1.1 Les maillages progressifs (*Progressive Mesh*)

Une technique de CLOD très répandue s'appelle *Progressive Mesh* [Hop96], elle s'applique sur des maillages quelconques et permet grâce à une structure de données adaptée de choisir dynamiquement le nombre de triangles d'un maillages par régions. Cette technique utilise une série d'opérations d' « affaissement de face » (*edge collapse*) appliquées récursivement ce qui permet ainsi de passer d'un maillage très détaillé à un unique triangle. Le problème majeur de cette technique est que la structure de données utilisée est très coûteuse en espace mémoire. Elle interdit également le stripping de triangles et impose un certain nombre de restrictions sur beaucoup d'opérations et optimisations réalisables par le matériel graphique.

1.2 L'algorithme de Röttger et ROAM

Ces deux techniques sont assez proches et se basent comme d'autres encore que nous n'avons pas étudiées (Lindstrom-Koller etc.) sur une estimation d'erreur en espace image et une génération à partir d'une structure d'arbre (binary-tree, quadtree, diamond…). Au contraire des *Progressive Mesh*, ces techniques s'appliquent uniquement sur des maillages réguliers plans de type table de hauteur. Ils permettent une génération très précise du maillage en détaillant exactement les zones qui le nécessitent le plus. Leur problème majeur est qu'ils nécessitent une regénération totale du maillage à chaque changement de point de vue. Aucune donnée ne peut donc être cachée sur le matériel graphique et l'ensemble des sommets doivent ainsi être retransmis à chaque image. Ces techniques semblent par contre bien s'appliquer aux rendus distribués. [Rot98], [Duc97], [GMMN05]

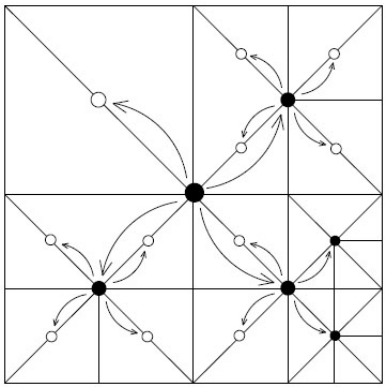


Illustration de l'algorithme de Rottger avec la représentation des différents nœuds de l'arbre de parcours

1.3 Le MipMapping Géométrique et les techniques tournées vers le matériel

Toutes les techniques présentées précédemment ont été inventées longtemps avant l'arrivée du matériel graphique moderne et des cartes 3D grand public telles que nous les connaissons actuellement. Elles ont donc été inventées dans l'unique optique de réduire le nombre de polygones affichées et ne prennent pas du tout en compte les contraintes et optimisations liées à ces architectures. Elles travaillent en effet triangle par triangle sans aucun regroupement ou traitement possible par batch (elles permettent au mieux l'utilisation de petits triangle FAN).

La technique que nous avons adoptée et mise en œuvre dans ce projet s'appelle « *Geometrical Mipmapping* » et a été introduite en 2000 par Willem H. Boer [DeBoer00]. Il s'agit donc d'une technique relativement récente donc et qui a été développée dans l'optique d'une bonne utilisation du matériel graphique qui commençait déjà en 2000 à proposer des capacités intéressantes. Le principe de cette technique et l'implémentation que nous en avons faite seront détaillées dans la partie suivante.

Une autre technique appelée « *Geometry Clipmap* » qui poursuit sur cette voie de l'utilisation du matériel graphique a attiré notre attention [LoHo04]. Cette technique présentée en 2004 semble être l'aboutissement de l'évolution des techniques de simplifications allant de plus en plus vers la génération de maillages très réguliers permettant une mise en cache aisée sur le GPU ainsi qu'une connectivité très régulière entre les zones de simplification.

Le problème majeur de tous les algorithmes précédents était en effet leur utilisation importante du bus avec la carte 3D. Etant entièrement mis en œuvre sur le CPU, ils nécessitaient d'importants transferts de données vers le matériel graphique à chaque image.

Cette technique se base donc sur l'alimentation maximum du pipeline graphique à chaque image en fixant un nombre de triangles rendus et en les répartissant simplement au mieux en fonction de la distance au point de vue. Des « anneaux » rectangulaires contenant chacun un niveau de détail sont ainsi dessinés autour de l'utilisateur et à chaque déplacement, seules les données devenues nécessaires sont transmises à la carte pour compléter chaque anneau.

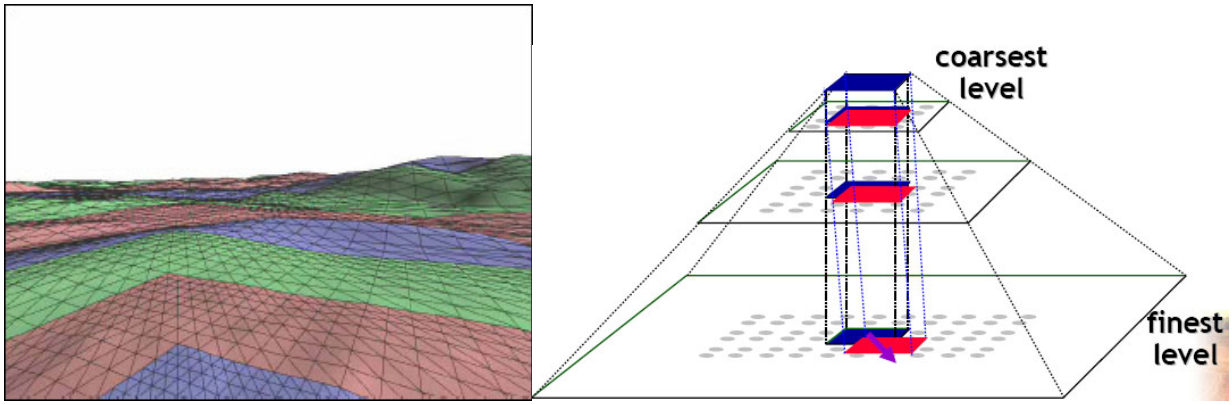


Illustration de la technique de « Geometry Clipmap » avec un aperçu de la série d'anneaux rectangulaires centrés sur le point de vue ainsi que le modèle en pyramide qui illustre l'application des données à partir des différents niveaux de détail sur les grilles imbriquées formants les anneaux.

Le problème de cette technique est qu'elle suppose que l'utilisateur se déplace de manière continue et linéaire sur le terrain et elle permet difficilement l'observation globale en hauteur.

Elle semble également difficile à étendre à de vastes étendues de terrain principalement du fait de l'occupation mémoire importante qu'elle nécessite due à l'imbrication des zones et à la non prise en compte du relief (seule la distance au point de vue est prise en compte).

C'est pour ces raisons que nous nous sommes plutôt tourné vers le « *Geometrical Mipmapping* » qui semblait mieux répondre à nos critères et offrir une plus grande souplesse dans l'implémentation et les extensions possibles (Très gros volumes de données, vision globale, Out-of-core, etc.).

2. Algorithme de visualisation

2.1 Présentation

La technique de rendu adaptatif que nous avons mise en œuvre se base intégralement sur la méthode exposée dans l'article de *De Boer* qu'il a donc fallu dans un premier temps comprendre et interpréter et à laquelle nous avons apporté nos propres améliorations. Il a également fallu inventer nos propres algorithmes afin de combler certains manques ou aspects non détaillés de l'article qui expose le principe général de l'algorithme en éludant parfois certains points techniques, ou détails d'implémentation, qui ne sont pas toujours si évident à mettre en oeuvre. Nous avons également développé un certain nombre d'optimisations principalement par rapport à l'utilisation du matériel graphique qui est un aspect totalement absent de l'article (qui date tout de même de 2000, une époque ou un certain nombre de fonctionnalités maintenant disponibles n'existaient pas).

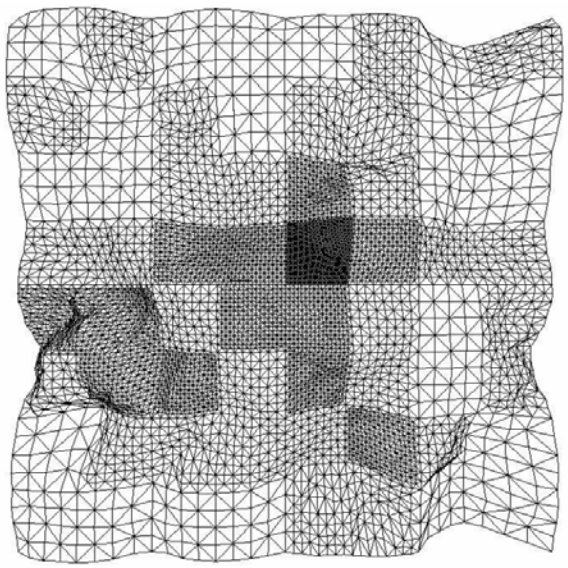
Un important travail a également été réalisé sur les structures de données et sur l'architecture mise en œuvre (cf. *chap 4.*) afin de fournir des objets suffisamment génériques pour pouvoir être réutilisés dans un contexte de visualisation plus large mais également afin de garantir l'évolutivité de l'algorithme.

2.2 Principe

Le principe du « *Geometrical Mipmapping* » est assez semblable à celui du mipmapping de textures. Il s'agit de découper le terrain en un certain nombre de « blocs » (ou *patchs*) de tailles identiques et pour lesquels on génère une série de maillages pour chaque niveau de détail. Chaque bloc doit avoir une résolution de $(2^n+1)x(2^n+1)$ permettant de générer n+1 niveaux de détails. Le niveau 0 correspond ainsi au maillage le plus détaillé et chaque niveau suivant supprime une ligne et une colonne sur deux du maillage précédent.

Au moment de l'affichage, le niveau le plus approprié est choisi pour chaque bloc en fonction de la distance au point de vue et du relief du terrain afin de minimiser le nombre de triangles rendus tout en conservant suffisamment de polygones sur les zones les plus détaillées. Pour cela, on fixe un seuil d'erreur maximal en espace image permettant d'évaluer la perte de qualité visible par rapport à l'affichage du niveau le plus détaillé.

Afin de contrôler la précision du rendu, il est donc possible de jouer sur deux paramètres : le seuil d'erreur en espace image ainsi que la taille des blocs générés.



Exemple vue de dessus d'un terrain décomposé en une série de patchs de résolution différentes

2.3 Génération des blocs

2.3.1 Présentation

La *figure 1* présente l'exemple d'un bloc de 5x5 dans sa résolution maximale (niveau 0), les rond noirs représentent les sommets qui formeront le niveau détail suivant (niveau 1), le 3ieme et dernier niveau de détail sera uniquement constitué des 4 sommets aux coins du maillage.

Les données du terrain étant issues d'une table de hauteurs, à chaque bloc est affecté une portion (de taille $(2^n+1)x(2^n+1)$) de la table dont il est chargé du rendu. Le niveau 0 associe ainsi un sommet à chaque élément de la table de hauteur, chaque bloc partageant ses bordures avec ses 4 blocs adjacents (cf. *Fig. 2*).

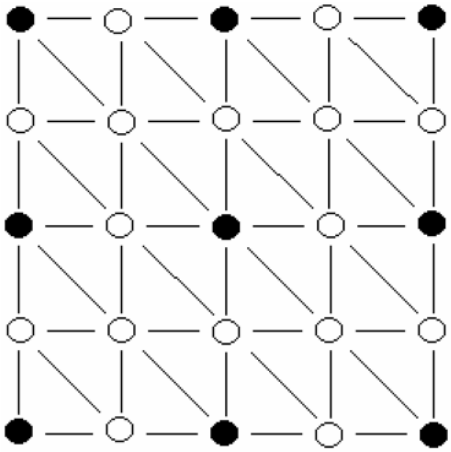


Fig.1 : Exemple d'un bloc de 5x5

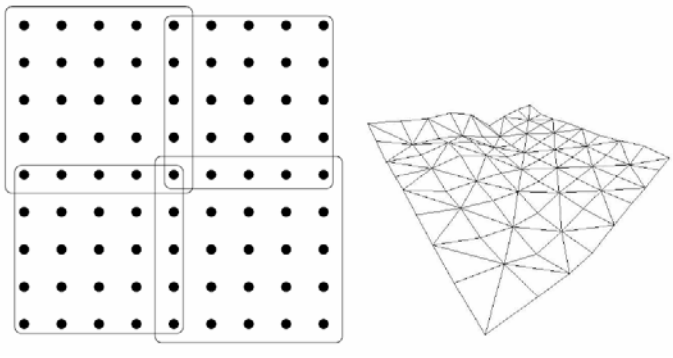


Fig2 : Illustration du partage d'une table de hauteur en 4 blocs

2.3.2 Maillage et connexion des blocs

Failles géométriques

L'un des aspects les plus critiques de l'algorithme a été la gestion des connections entre les blocs de niveau de LOD différents. Le maillage de chaque bloc étant généré indépendamment, il faut en effet pouvoir assurer la continuité globale du maillage qui sans cela serait parsemé de brèches aux jonctions entre blocs. Cette gestion doit en en plus être effectuée dynamiquement, le niveau de détail d'un bloc ou de l'un de ses voisins pouvant changer à chaque instant.

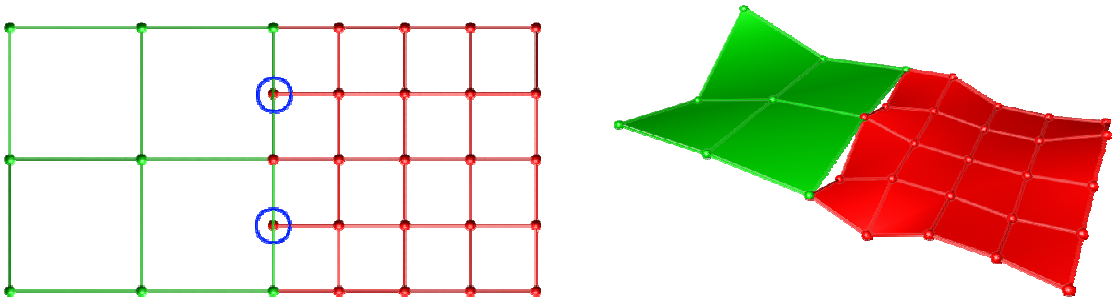


Fig. 3 Illustration du problème de brèches aux jonctions de blocs

Pour combler ces brèches, 3 solutions sont possibles :

- Déplacer les sommets du maillage le plus détaillé afin qu'ils viennent se positionner à bonne hauteur sur l'arrête du maillage le moins détaillé (position médiane entre les deux sommets adjacents le long de l'arrête dans le cas d'une différence de 1 des niveaux de détail).
- Créer de nouveaux sommets sur l'arrête du maillage le moins détaillé afin de venir s'aligner sur les sommets supplémentaires du maillage le plus détaillé.

- Supprimer les sommets de l'arrête du maillage le plus détaillé ne correspondant à aucun sommet dans le maillage le moins détaillé.

C'est cette dernière solution que l'article de *De Boer* propose d'utiliser. Elle a en effet l'avantage de ne pas nécessiter l'utilisation d'espace mémoire supplémentaire comme c'est le cas pour la 2ième solution. Elle permet également d'éviter le problème des « jonctions en T » (*T-junction*) qu'entraîne la 1ere solution. Ce phénomène bien connu en infographie est dû à l'imprécision de calcul sur les flottants lors de l'évaluation de la position du sommet du coté du maillage le plus détaillé pour venir se conformer à l'arrête du maillage le moins détaillé. Cette imprécision fait que les deux sommets ne sont pas positionnés exactement au même endroit ce qui entraîne l'apparition d'artefacts le long des arrêtes des triangles ainsi juxtaposés.

Mise en œuvre proposée

Pour mettre cette stratégie en œuvre, l'article suggère de gérer de manière indépendant le maillage des bordures en connexion avec un bloc de résolution différente de celui du reste du bloc. Implicitement, l'auteur propose de mailler la partie régulière de chaque bloc à l'aide de triangle STRIP bien adaptés à la modélisation de longues bandes de triangles et qui permettent déjà de *batcher* un minimum le rendu. Pour gérer les bordures dans le maillage le plus détaillé, il suggère de dessiner une série de triangle FAN permettant de relier les sommets présents dans le maillage adjacent à ceux du maillage interne du bloc comme illustré *figure 4*.

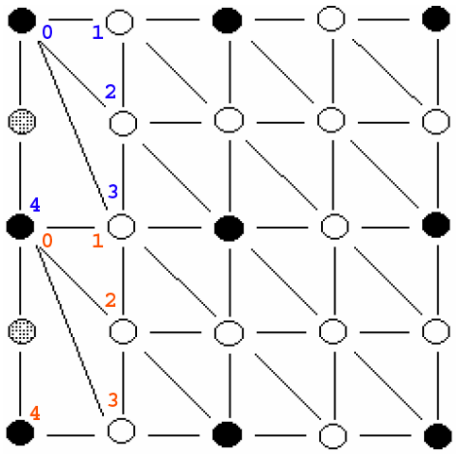


Fig. 4 Illustration de l'utilisation de 2 triangle FAN pour le maillage d'une bordure en contact à gauche avec un bloc de résolution inférieure (niveau de LOD +1)

Problèmes soulevés et utilisation du matériel

Globalement, une gestion des connexions introduit une dépendance de chaque bloc vis-à-vis de ses voisins ce qui fait perdre une bonne partie de l'intérêt de l'algorithme dans son utilisation avec du matériel graphique. L'idée est en effet de limiter au maximum les transferts vers la carte en cachant les données de sommets en mémoire graphique par l'intermédiaire de VBO [GLR00] statiques (*GL_STATIC_DRAW*). Ces données doivent donc être modifiées le moins souvent possible pour que l'algorithme soit efficace.

Il est également intéressant de pouvoir manipuler de gros morceaux de maillages plutôt que de multiples petits afin de limiter le nombre d'appels aux primitives OpenGL de dessin de tableaux de vertex (*Vertex Array*). Ces appels peuvent en effet constituer une source de ralentissements substantiels et limiter leur nombre peut être un bon moyen pour optimiser un algorithme de rendu.

Dans une implémentation naïve de la méthode de *De Boer*, utilisant un tableau de sommets et une série de primitives indexées sur ce tableau (*Vertex Array*), le dessin de l'ensemble des triangle STRIP par bandes et celui des triangle FAN pour la gestion des bordures s'avère relativement conséquent (cf. *fig. 5*). Une première amélioration consiste à utiliser la capacité du matériel graphique à éliminer les triangles dégénérés pour « bricoler » un seul long triangle STRIP permettant de dessiner toute la partie régulière d'un bloc avec un seul appel à la primitive de dessin (plutôt que de faire un appel par bande). Mais même avec cette solution, il reste toujours à dessiner les différents triangles FAN des bordures ce qui coûte encore un certain nombre d'appels déparés de dessin de primitives.

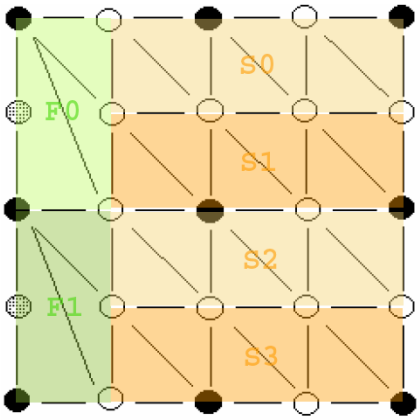


Fig.5 Illustration du maillage naïf d'un bloc avec gestion d'une connexion à gauche par la méthode décrite dans l'article de De Boer. En vert les triangle FAN, en orange les triangle STRIP.

2.3.3 Maillage mis en oeuvre

Nous avons donc cherché à mettre en œuvre un algorithme de rendu de maillage qui réponde mieux aux contraintes d'utilisation de la carte 3D en cachant un maximum de données sur la carte et en limitant le rendu d'un bloc à une unique primitive de dessin.

Pour cela, nous avons eu recourt à une extension proposée par nVidia et disponible sur l'ensemble de sa gamme de produits. Appelée *NV_primitive_restart* [GLR00], cette extension permet à l'aide d'un indice spécial dont la valeur est paramétrable, de redémarrer une primitive indexée au cours de son tracé. Cela permet ainsi de tracer de multiples primitives du même type (des triangle STRIP par exemple) à l'aide d'un seul appel à `glDrawElements`.

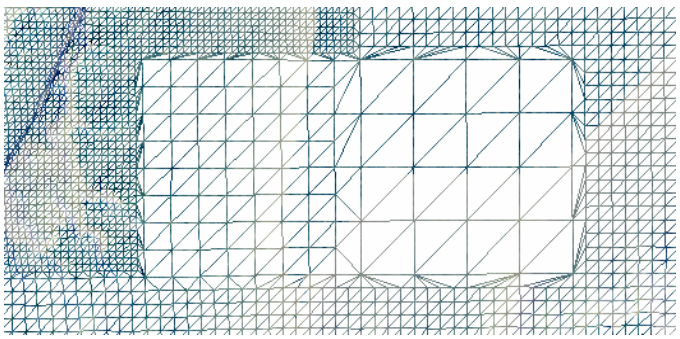


Fig.6 Illustration du maillage et de la connexion avec les blocs voisins mis en œuvre.

Tableau de sommets

Les données de sommets (position, coordonnées de mapping et informations de morphing, cf. *2.5*) étant statiques et communes à l'ensemble des niveaux de mipmapping, il est possible de les conserver de manière unique pour chaque bloc en mémoire graphique. L'ensemble des niveaux est donc rendu par indexage d'un même tableau de sommets généré une fois pour le

niveau le plus détaillé à partir de la table de hauteur. Ce tableau une fois généré est stocké en mémoire graphique (ou en mémoire AGP rapidement accessible par la carte) sous la forme d'un VBO statique (*GL_STATIC_DRAW*).

Ce VBO est ensuite utilisé au travers des fonctions de *Vertex Array* pour spécifier les composantes de position, de coordonnées de texture et les paramètres de morphing (sous forme également de coordonnées de texture).

Pour une efficacité maximale, nous avons veillé à utiliser une structure de Vertex alignée sur 32 octets, c'est-à-dire dont la taille est un multiple de 32 octets (notre structure fait exactement 32 octets). Ceci permet (sur du matériel nVidia en tout cas) d'améliorer l'efficacité de transfert et de manipulation du tableau en évitant que le driver OpenGL ait à réorganiser les données pour les envoyer à la carte.

Maillage

Le rendu de chaque niveau se fait ensuite en indexant uniquement les sommets nécessaires au travers d'un unique triangle STRIP. L'algorithme de maillage que nous avons créé a pour but de permettre une prise en charge des connexions avec les blocs adjacents comme expliqué précédemment d'une manière simple et à partir de l'unique primitive de dessin. Il permet de gérer les changements de résolution des blocs adjacents avec un minimum de modifications dans le tableau d'indices.

Comme illustré *figure 7*, le principe est de générer le maillage (tableau d'indices) par bandes de triangles STRIP horizontales terminées par un indice de remise à zéro défini grâce à *NV_primitive_restart* (en prenant garde à l'orientation des faces pour le *back face culling*):

```
//stepX, stepY : Pas horizontal et vertical en fonction du niveau (2^niveau)
//size : Résolution totale de la matrice de sommets
Ind=0;
for(j=sY-1; j>0; j--){
   for(i=0; i<sX; i++){
         Indices[ind++]= ((i*stepX)   + (j*stepY)   *size.x);
         Indices[ind++]= ((i*stepX)   + ((j-1)*stepY) *size.x);
   }
   Indices[ind++] = RESTART_PRIMITIVE_CST;
}
```

A partir de ce maillage de base, la prise en charge des connexions avec les blocs voisins se fait par dégénération de triangles. Les indices de chaque bordure en connexion avec un bloc de résolution différente sont ainsi parcourus et si le sommet dessiné n'a pas de correspondant dans le bloc adjacent, l'indice du sommet le plus proche sur la bordure et ayant un correspondant est placé en remplacement. Le maillage étant dessiné par bandes, cette règle simple a pour effet de générer les triangles corrects pour assurer l'ensemble des connexions possibles et quelque soit la différence de niveau entre les blocs.

Cette technique a l'avantage de ne pas modifier la taille du tableau d'indices et d'intervenir uniquement localement sur les points à traiter à chaque changement de résolution d'un voisin. Ceci permet part exemple de stocker des tableaux d'indices, par niveaux, communs à l'ensemble des blocs et qui sont modifiés de manière rapide et localisée à l'affichage de chaque bloc. Les différents tests que nous avons réalisés ont montrés qu'il était plus intéressant de conserver les tableaux d'indices en mémoire centrale et de les envoyer en bloc lors du rendu plutôt que de les stocker sous forme de VBO que l'on *map* pour modifier les bordures avant chaque rendu. Cela permet en plus d'économiser de la place en mémoire graphique.

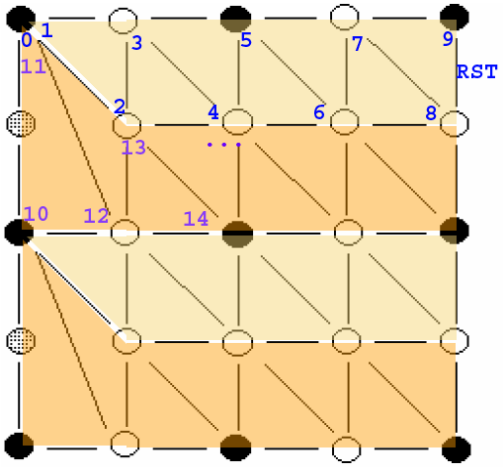


Fig. 7 Illustration de notre algorithme de maillage pour un niveau de LOD d'une résolution de 5x5. Les numéros représentent l'ordre d'utilisation des sommets dans le tableau d'indices de triangle STRIP.

2.3.4 Structures de données

La notion de bloc a été abstraite sous la forme d'une classe *TerrainBloc* (cf. *fig. 7*) permettant d'encapsuler l'ensemble des données et traitements nécessaires à la création et à la visualisation d'un bloc. Un bloc est constitué d'un ensemble de niveaux matérialisés par la classe *LODLevel* (cf. *fig 8*) qui permet de stocker les données relatives à un niveau de *mipmap*.

L'avantage de cette structure est quelle permet un partage flexible des données entre les niveaux. Cela fournit ainsi une bonne plateforme d'expérimentation afin de tester les configurations les plus efficaces et a permis de trouver le meilleur compromis entre occupation mémoire et efficacité du code. Le partage des données de sommets entre niveaux est ainsi possible mais pas obligatoire et la manipulation de données mappées à partir d'un VBO peut également être abstraite et utilisée de manière transparente au niveau des algorithmes.

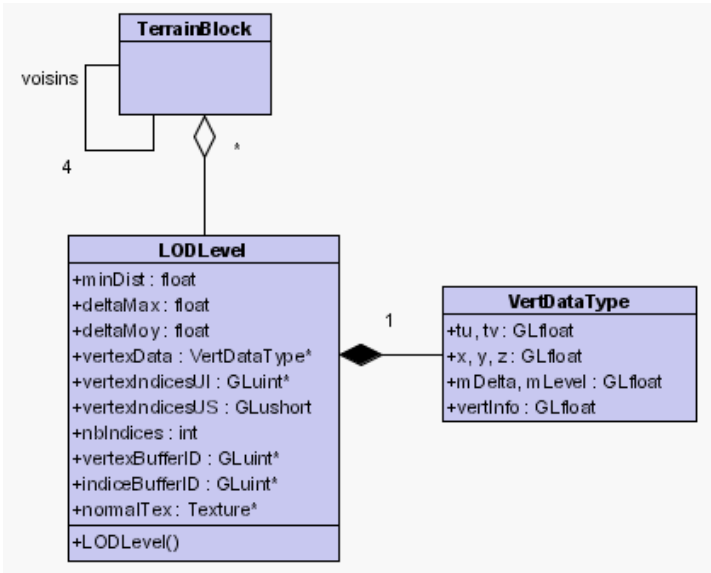


Fig.8 Diagramme de la classe LODLevel

2.4 Choix du niveau de mipmap

2.4.1 Problématique

Comme expliqué précédemment, le choix du niveau de détail utilisé se fait à partir de l'évaluation d'un seuil d'erreur en espace image. En effet, un simple choix du niveau de détail en fonction d'une distance fixe au point de vue ne suffirait pas et produirait au moment du changement de niveau d'un bloc d'importants sauts et changements brusques plus ou moins visibles en fonction de l'angle d'observation. Comme pour le *mipmapping* de textures, le choix du niveau doit donc être fait en fonction du résultat en espace image.

Lors du passage d'un niveau de détail au niveau supérieur (de résolution inférieure donc, le niveau 0 étant le plus détaillé), le retrait d'un certain nombre de sommets entraîne en effet une erreur de hauteur δ en ces points comme illustré *figure 9*. L'erreur visuelle résultante est la projection ε de ce δ en espace image.

Chaque niveau comporte donc un ensemble de ces erreurs, une par sommet retiré du niveau inférieur. Le choix de passage ou non a un niveau se fait donc en comparant le maximum de ces ε avec l'erreur maximale τ autorisée. Il faut donc, à chaque changement de point de vue, réévaluer pour chaque bloc l'ensemble des ε générés pour chaque niveau et comparer leur maximum avec le τ autorisé afin de choisir le niveau le moins détaillé qu'il est possible d'utiliser.

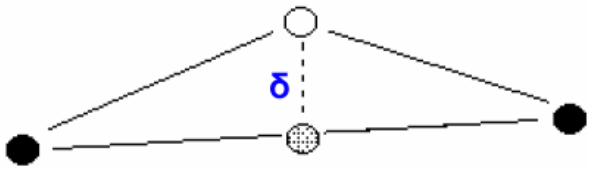


Fig. 9 : Illustration de l'erreur de hauteur issue du retrait d'un sommet (en blanc). La ligne pointillée représente le changement de hauteur qui en résulte

Cette méthode permet ainsi de choisir avec précision le niveau de détail minimum pour chaque bloc mais elle se révèle extrêmement coûteuse en calculs. La priorité lors de l'utilisation du matériel graphique étant d'alimenter le pipeline au maximum de ce qu'il peut traiter, il n'est pas primordial de choisir très précisément la résolution minimale pour chaque bloc. Il est ainsi préférable d'alléger les traitements à réaliser pour le choix d'un niveau quitte à rendre légèrement plus de polygones.

2.4.2 Optimisation

Le calcul de ε dépendant de la position du point de vue et de son orientation, il n'est pas possible de pré-calculer le ε maximum pour l'ensemble des niveaux d'un bloc mais en considérant une orientation du point de vue constamment horizontale, ce prétraitement devient possible. Lors du chargement d'un bloc, on pré-calcule alors une distance minimum d pour chaque niveau à partir du type de projection utilisée et du τ d'erreur maximum. Le choix du niveau lors du rendu se fait alors simplement en comparant la distance courante au point de vue au d de chaque niveau.

Pour le pré-calcul de d, l'article de *De Boer* propose une formule de simplification qui permet d'obtenir la distance minimum pour un niveau à partir du δ maximum, de la position du *near clipping plane* définissant la projection, de celle du *top*, de la résolution maximale de l'écran ainsi que du τ :

$D_n = |\delta| \cdot C$, avec δ erreur maximum du niveau

et $C = \frac{A}{T}$, où

$A = \frac{n}{|t|}$, avec n coordonnée du *near clipping plane*, t coordonnée du top *clipping plane*

et $T = \frac{2 \cdot \tau}{v_{res}}$, avec τ erreur maximum autorisée et *Vres* résolution verticale de rendu.

2.4.3 Mise en œuvre

Pour chaque niveau de *mipmap*, il est nécessaire de calculer δ uniquement pour les sommets ayant disparus par rapport au niveau inférieur. Lors du passage d'un niveau à un autre, les sommets disparaissent toujours entre deux autres sommets qui se connectent pour former une nouvelle arrête. Le calcul de l'erreur se ramène donc à la différence de hauteur entre le sommet anciennement présent et le milieu de l'arrête nouvellement formée (cf. *fig 10*), la hauteur du milieu de l'arrête étant simplement calculée par interpolation linéaire.

Pour un niveau donné, le calcul de l'erreur maximum a été fait en parcourant chaque sommet (sauf le dernier de chaque ligne et colonne) et en prenant en compte le rectangle issu de ce sommet (cf *fig. 10*) pour calculer les 3 erreurs à l'intérieur de ce rectangle. Un cas particulier est traité pour les derniers rectangles de lignes et de colonnes afin de calculer les 5 erreurs.

En plus d'être utilisées pour la détermination de l'erreur maximum par niveau, ces erreurs sont stockées par sommets afin d'être utilisées dans l'algorithme de *géomorphing* détaillé dans le chapitre suivant.

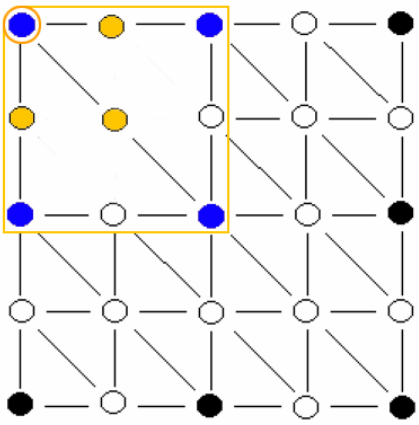


Fig. 10 : Illustration de l'algorithme de calcul d'erreur pour un niveau de mipmap (sommets noirs et bleus)

2.5 Géomorphing

2.5.1 Présentation

Bien que l'algorithme de rendu tel qu'il vient d'être présenté fournisse déjà de très bons résultats, en fixant un taux d'erreur acceptable à l'écran (τ) suffisamment bas (une valeur de 5 pixels semble suffisante dans une majorité de cas), une bonne façon d'améliorer la qualité du rendu sans ajouter de polygones est d'implémenter une forme de morphing pour le passage d'un niveau de *mipmap* à un autre. Ceci permet d'effectuer une transition progressive entre les maillages qui élimine le phénomène d'apparition et de disparition brusque de polygones et permet ainsi d'augmenter le taux d'erreur à l'écran sans perte de qualité visuelle.

L'intérêt de cette technique est qu'elle peut être mise en œuvre entièrement sur le matériel graphique moderne, dans un programme de traitement de sommets, pour un coût quasiment nul. Il s'agit en fait de l'équivalent géométrique du filtrage trilinéaire du *mipmapping* de textures.

2.5.2 Principe

Le principe du *géomorphing* est de considérer en plus du niveau de *mipmap* courant un taux de passage entre deux niveaux. Ce taux de passage entre 0.0 et 1.0 est calculé à partir de la distance courante au point de vue et en fonction de la distance minimum du niveau courant et celle du niveau supérieur (de résolution inférieure donc). Il permet d'interpoler pour le niveau courant la position (hauteur) des sommets qui disparaîtrons dans le niveau supérieur, entre leur position réelle et la position qu'ils doivent prendre pour venir se conformer à l'arrête qui sera créée lors de leur disparition (cf. le point gris de la *figure 9*).

2.5.3 Mise en œuvre

Ce morphing a été implémenté dans un *vertex shader* et il a donc fallu ajouter aux sommets les informations nécessaires à l'interpolation. Le propre d'un programme de traitement de sommet est d'appliquer un traitement identique à l'ensemble des sommets dessinés. Seulement lors du dessin d'un niveau de *mipmap* donné, tous les sommets dessinés n'ont pas besoin d'être interpolés. Si l'on regarde d'un peu plus près le comportement des sommets, on s'aperçoit qu'en fait chaque sommet du maillage complet (niveau 0) ne morphe qu'une seule fois dans un seul niveau de mipmap.

Afin de conserver la structure de rendu telle qu'elle a été présentée précédemment, c'est-à-dire avec un tableau de sommets statique partagé par l'ensemble des niveaux de *mipmap*, nous avons ajouté deux informations à chaque sommet : le δ d'erreur calculé pour le choix du niveau de *mipmap* et qui permet à partir de la hauteur du sommet (position en z) de trouver la hauteur vers laquelle il doit *morpher*, ainsi que le niveau au cours duquel le morphing doit intervenir.

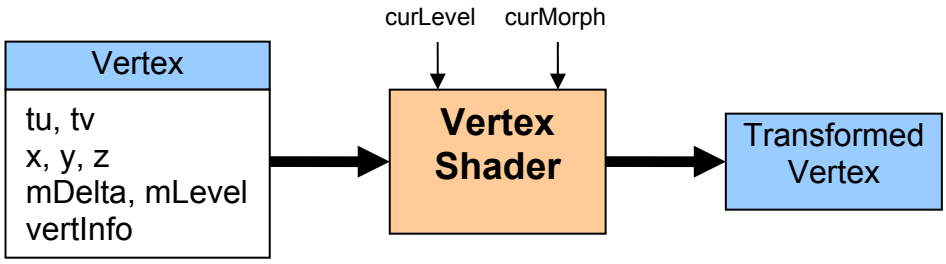


Fig.11 : Schéma de principe du vertex shader mis en oeuvre

Le *shader* prend donc en paramètre global le niveau de *mipmap* courant ainsi que le taux de morphing et calcul la position interpolée des sommets concernés par un morphing au niveau courant. La formule d'interpolation est finalement très simple :

$$hVertex = vertex.z - (curMorph \cdot vertex.mDelta)$$

2.6 Frustum culling

2.6.1 Présentation

Lors de l'exploration d'une scène, une grande partie de celle-ci n'est souvent pas visible car en dehors de la zone de vision. Il n'est donc pas nécessaire de rendre les objets situés en dehors de cette zone et cela permet d'obtenir un gain substantiel de performances. Cette élimination des parties invisibles s'appelle Frustum Culling et est mis en œuvre couramment en infographie à l'aide d'algorithmes de partitionnement de l'espace.

La structure des terrains rendus étant intrinsèquement bidimensionnelle (tables de hauteur donc pas de recouvrement sur la dernière dimension), l'algorithme de partitionnement que nous avons mis en place se base sur une structure de *Quadtree*. Ce type de structure est un arbre de partitionnement de l'espace qui permet de stocker les objets à visualiser et d'éliminer rapidement, grâce à son organisation hiérarchique, des zones de l'espace en fonction d'un certain critère.

Le *Quadtree* est un arbre dont chaque nœud possède 4 fils et est couramment utilisé pour le rendu de terrains. Chaque nœud définit la boite englobante de la zone qu'il contient et les feuilles de cet arbre stockent les données à afficher. On définit ainsi une hiérarchie de boites englobantes que l'on soumet au test de visibilité au cour d'un parcours descendant de l'arbre. Le rejet d'un nœud haut dans la hiérarchie élimine ainsi d'un seul coup l'ensemble de ses fils qui ne seront forcément pas visibles. Bien que la structure soit plane, les boites englobantes sont bien entendu définies en 3D. Pour notre implémentation chaque feuille de l'arbre contient l'un des blocs du terrain dont on calcul la boite englobante.

Le test de visibilité est réalisé en calculant l'intersection du *frustum* (qui définit la pyramide de vue) avec la boite englobante de chaque nœud. Il faut pour cela calculer l'intersection de la boite avec les 6 faces de la pyramide.

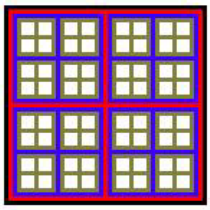


Fig.12 : Exemple de quadtree

2.6.2 Mise en œuvre

La notion de *quadtree* a été implémentée sous la forme d'une classe *QuadTree* totalement générique et permettant de manipuler tout type d'objet pouvant être rendu. Elle stocke en effet comme élément de ses feuilles, une série de références sur une interface *Renderable* qui définit l'ensemble des méthodes acceptables par les objets graphiques. Elle implémente également elle-même cette interface en toute logique ce qui permet diverses imbrications.

Le test de visibilité d'un *Renderable* est effectué au travers d'une méthode *isVisible* de la classe *Camera* qui sert également à définir et à manipuler un point de vue. Cette méthode utilise un objet *Frustum* dédié aux tests de visibilité.

L'arbre est construit de bas en haut à partir de la matrice de blocs préalablement construite et est manipulé simplement par appel à ses méthodes *updateGeometry* et *render* qu'il répercute sur les objets visibles qu'il contient.

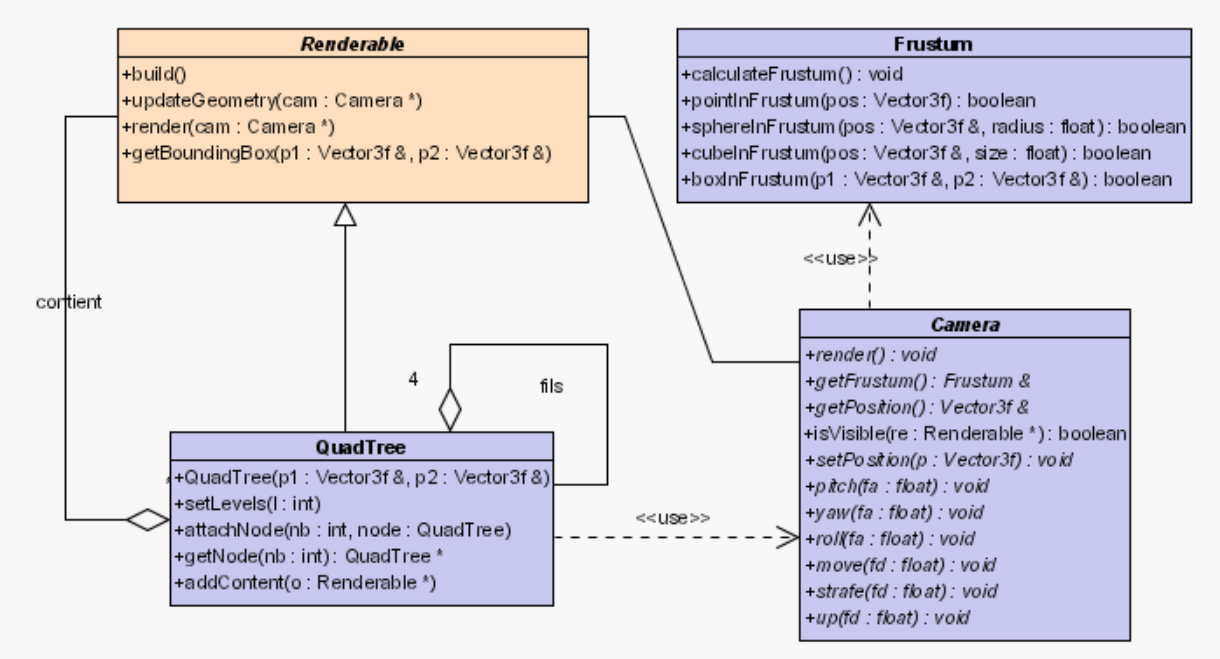


Fig. 13 : Diagramme de classe de la partie dédiée au Frustum Culling

2.7 Matériau et éclairage

2.7.1 Eclairage du modèle

Présentation

Afin d'ajouter d'avantage de réalisme au rendu, une gestion de l'éclairage a été mise en place sur le modèle. Il s'agit d'un éclairage dynamique par fragment qui permet, quelque soit la résolution du maillage rendu, de conserver la prise en compte de tous les détails de la surface. Pour cela, une table de normales (*normal map*) est générée à partir de la table de hauteurs et gérée par blocs. Cette table est ensuite transmise sous la forme d'une texture RGBA à la carte 3D pour être utilisée dans un programme de traitement de fragments afin de calculer les composantes d'éclairement pour chaque fragment. La gestion par bloc permet un traitement parallèle à celui des sommets pour la simplification et le *mipmapping* de ces textures.

Génération de la table de normales

Les normales au terrain sont donc estimées en tout point de la table de hauteur. Cette estimation est faite à partir des différences de hauteur autour du point considéré sur chaque axe comme illustré *figure 14*. Le calcul se fait donc en deux passes dans lesquelles ont détermines la droite reliant les deux voisins sur l'axe considéré puis l'on en déduit la normale dans le plan considéré. Les deux normales sont ensuite ajoutées et le vecteur obtenu est normalisé avant d'être inséré dans la table.

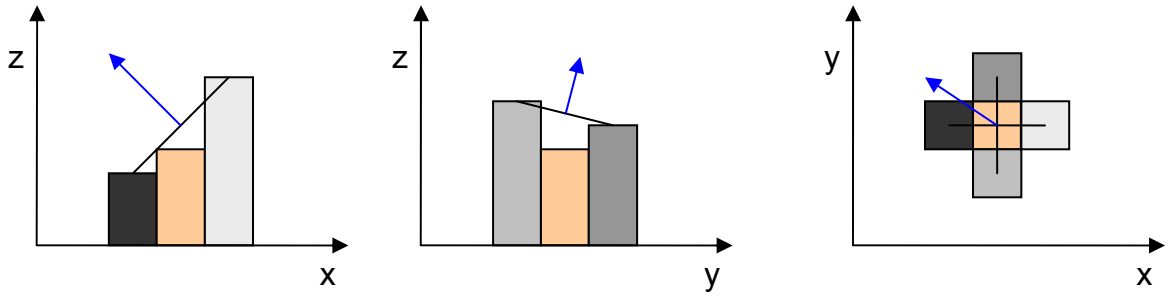


Fig.14 : Estimation de la normale en un point (en orange) de la table de hauteur.

Calcul de l'éclairage

La table ainsi générée est ensuite transformée en texture 32bits dans laquelle, pour chaque pixel, on place dans les composantes RGB les composantes x, y et z du vecteur normal correspondant et dans Alpha, la hauteur du point (elle sera utilisée pour la génération de la texture paramétrique). L'extension *ARB_texture_rectangle* a été utilisée afin de pouvoir manipuler des textures dont les dimensions sont quelconques. Dans le *fragment program*, le modèle *de phong* [PHON75] est utilisé pour mettre en œuvre un *bump mapping* à partir de la *normal map*.

2.7.2 Texturage

L'application des textures de matériaux (Herbe, roche, sable…) sur le maillage a été fait de manière paramétrique à partir de textures statiques. Une texture de chaque matériau est passée au programme de traitement de fragments qui les combine en fonction de la hauteur et de l'inclinaison du terrain, récupérés dans la *normal map*. L'intérêt d'utiliser la *normal map* plutôt que des données issues des sommets, outre la précision supérieure obtenue, est qu'elle permet une indépendance vis-à-vis de maillage et assure ainsi une apparence constance, sans variation, quelque soit le niveau de simplification. Cela, combiné au calcul d'éclairage, permet d'obtenir un rendu de terrain qui reste extrêmement réaliste quelque soit la distance d'observation et le taux de simplification des blocs.

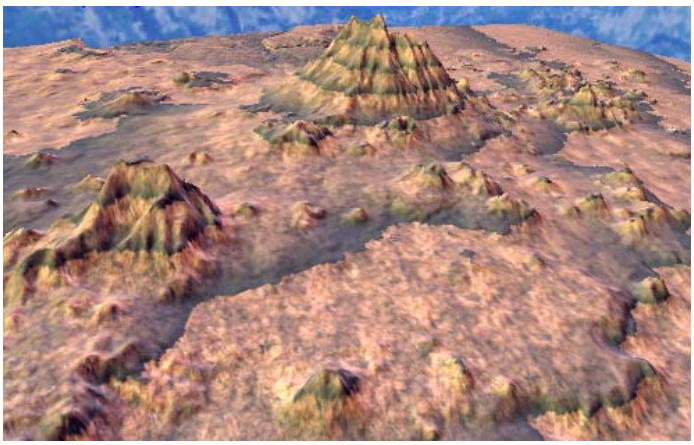


Fig.15 : Illustration du matériau paramétrique mis en place

3. Gestion des tables de hauteurs

3.1 Présentation

La table de hauteur est la source de données pour la génération du terrain. Nous avons abstrait sa gestion dans une classe nommée *HeightMap* qui fournit un ensemble de méthodes de chargement à partir de divers formats de fichiers images. Elle propose également toutes les interfaces d'accès à ses données, de localisation spatiale et de génération de tables de hauteurs. C'est au travers de cette interface que l'ensemble des algorithmes présentés récupèrent leurs données.

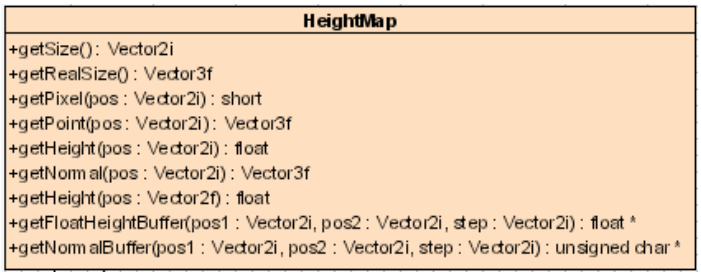


Fig.16 : Classe HeightMap

3.2 Les fichiers HGT

3.2.1 Présentation

Au cours de notre recherche de tables de hauteurs pouvant servir d'exemple pour tester notre moteur, nous sommes tombé sur une énorme base de données gérée par la NASA contenant les cartes d'élévation de quasiment l'ensemble de la surface de la planète. Ces données sont proposées en téléchargement libre au travers d'un FTP publique [ECS00] sous la forme de fichiers HGT. Ces fichiers sont de deux types : des fichiers SRTM-1 d'une résolution de 3601x3601 points échantillonnés à 1 arc-seconde et correspondant à peu près à un rectangle de 125Km de coté ainsi que des fichiers SRTM-3 d'une résolution de 1201x1201 correspondant à la même superficie mais échantillonné à 3 arc-secondes. Il s'agit de fichiers binaires contenant des données brutes de hauteur codées sur 2 octets et organisés au format « big endian » (Motorola, les octets sont intervertit par rapport au format utilisé sur x86 et il a donc fallu les intervertir lors du chargement).

Ces fichiers nommés en fonction de leur latitude et longitude sous la forme « N15W079.hgt » qui signifie 15° Nord de latitude et 79° Ouest de longitude.

3.2.2 Classe de manipulation

Pour gérer ce type de fichiers, nous avons développé une classe *HeightMapHGT* qui permet de charger et de manipuler un ensemble de ces fichiers formant une zone contiguë de manière globale et transparente comme une unique table de hauteurs. Ce type de table est en effet constitué d'un ensemble de parties liées chacune à l'un des fichiers de la zone. Le chargement et le déchargement de chacune de ces parties peut être géré indépendamment ce qui permet de manipuler de très grandes surfaces et donc de très gros volumes de données sans charger l'ensemble en mémoire.

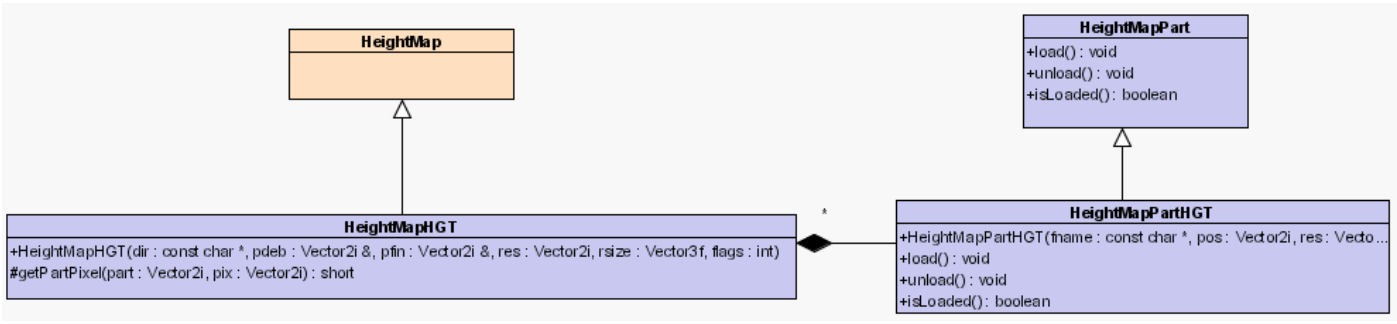


Fig.17 : Diagramme de classe de la gestion des fichiers HGT

4. Structure de l'application

L'ensemble de l'application a été développée sur un modèle entièrement objet avec un soucis constant de généricité et d'implémentation d'outils réutilisables. Les types de données constamment manipulés en infographie que sont les vecteurs à 2, 3 ou 4 composantes ont été implémentés avec un grand nombre de méthodes de manipulation et d'opérations ce qui s'est révélé extrêmement utile pour la clarté et la facilité d'implémentation des algorithmes. Ceci permet également de pouvoir passer facilement des points, des vecteurs, des coordonnées ou diverses informations de tailles de manière compacte entre méthodes.

Nous avons tenté de fixer une structure commune et un mode de fonctionnement unifié pour que les modules mis en place communiquent et fonctionnent ensemble de manière cohérente. Une interface *Renderable* a par exemple été définit afin de fixer les opérations disponibles sur l'ensemble des objets de rendu et permettre d'implémenter des conteneurs génériques tels que la classe *Quadtree*.

Nous nous sommes ainsi constitué une « boite à outils » ou un mini moteur de rendu avec des classes de manipulation des *shaders* par exemple utilisés pour le rendu des blocs au travers d'une interface générique qui permet d'abstraire le langage de définition de *shaders* utilisé. L'implémentation proposée ici et utilisée pour le projet est le Cg de nVidia mais il serait facile de rajouter la prise en charge d'autres langages. Une classe de gestion des textures a également été développée afin de pouvoir s'abstraire des opérations de chargement, des formats, des modes de filtrage etc.

La gestion globale d'une application avec l'abstraction du système de fenêtrage et des entrées sorties a également été mise en place au travers d'une classe GLApp. Celle classe implémente cette gestion au travers de la librairie GLUT (ou OpenGLUT) et est donc comme l'ensemble des modules développés totalement portable. Elle gère également tout ce qui concerne les extensions OpenGL grâce à la librairie GLEW.

Notre application de rendu de terrain appelée *GLAppTerrain* hérite donc de cette classe GLApp et redéfinit les méthodes d'initialisation, de rendu et de gestion du clavier et de la souris pour définir son comportement propre.

Un diagramme de classe complet de l'ensemble de l'application est disponible en *annexe 1*.

Bilan et perspectives

Ce projet d'IN55 nous a permis tout d'abord de mener de A à Z l'implémentation d'un algorithme de rendu relativement complexe, d'en appréhender les subtilités, d'en découvrir les faiblesses et d'inventer des solutions efficaces pour l'adapter aux architectures de rendu matériels modernes. La première étape de recherche bibliographique et de lecture de publications a également été très formatrice et les contraintes que nous nous étions imposées nous ont permises de nous fixer une ligne directrice relativement claire ainsi que des objectifs bien définis.

Les structures et algorithmes mis en place permettent déjà de visualiser de manière très efficaces de relativement grandes étendues de terrain. Nous avons en effet réussi à explorer une surface d'environ 400 000 Km^2, définie par une table de hauteur de 36 060 025 points (6005^2) répartie dans 25 fichiers de manière interactive. Ce test a été réalisé sous Linux sur un Athlon 64 3400+ équipé d'1 Go de RAM et d'une GeForce 6800 GT.

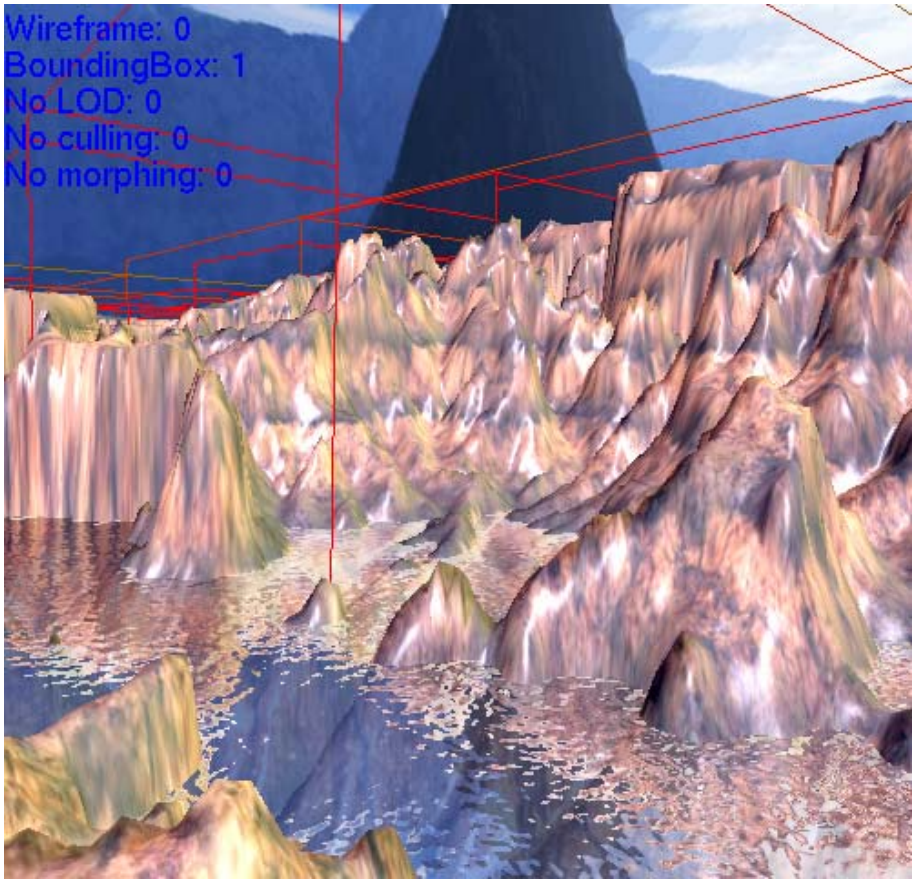
Le programme fonctionnant parfaitement sur les deux plateformes et *nVidia* fournissant des drivers Linux aussi performants que ceux destinés à Windows, nous avons pu comparer les performances du programme sur les deux environnements (nous développons sous Windows). Il s'est avéré que sur de gros terrains, le programme fonctionne environ 20% à 40% plus rapidement sous Linux que sous Windows. En poussant encore la taille des données manipulées, nous avons atteint les limites du mécanisme de mémoire virtuelle de Windows qui finit par interdire l'allocation de mémoire et interrompt le programme. Ce problème n'est pas présent sous Linux qui semble donc beaucoup mieux gérer sa mémoire virtuelle.

Actuellement, nous laissons en effet toute la gestion de la mémoire au système d'exploitation qui se charge de swapper sur le disque les pages mémoire le moins souvent utilisées et de les recharger en mémoire centrale lors de leur accès. Sur de très gros terrains, cette gestion résulte en une utilisation permanente du disque dur principalement lors de mouvements de rotation autour du point de vue ou de déplacements rapides sur le terrain ce qui crée d'importants ralentissements.

La gestion de la mémoire vidéo est également laissée au driver *OpenGL* et lorsque les données de sommets ou de textures deviennent trop importantes, un *swapping* identique est mis en œuvre entre la mémoire vidéo et la mémoire centrale.

Le support et l'exploration d'étendues de terrain encore plus importantes doit donc passer maintenant par une gestion explicite de ces deux mémoires, centrale et vidéo, au sein même du programme par l'intégration de mécanismes et d'algorithmes dit « *Out of core* ». Ce type de mécanisme permet de gérer des volumes de données ne tenant pas entièrement en mémoire. Cela se fait généralement en chargeant uniquement les données nécessaires à partir du disque, en désallouant les données lorsque leur probabilité d'utilisation devient faible et en anticipant le chargement des données qui deviendront nécessaires au cours de l'exploration.

Sur un terrain, cette gestion peut être faite de manière très géographique en comptant sur la cohérence des déplacements de l'utilisateur. L'ensemble des structures de données que nous avons mises en place, que ce soit au niveau des blocs ou des tables de hauteur, ont été développées dans l'optique de permettre et de faciliter la mise en place à terme de tels mécanismes.


Wireframe: 0
BoundingBox: 1
No LOD: 0
No culling: 0
No morphing: 0

Bibliographie

[Hop96] Hugues Hoppe. *Progressive meshes*. In Computer Graphics (proceedings of Siggraph '96), pages 99-108, 1996.

[Hop97] Hugues Hoppe. View-*dependent refinement of progressive meshes*. In Computer Graphics (proceedings of Siggraph '97), pages 189-198, 1997.

[Rot98] Röttger, S. and Heidrich, W. and Slusallek, P. and Seidel, H. P. *Real-Time Generation of Continuous levels of Detail for Height Fields*. Proceedings of WSCG '98, pages 315-322, 1998.

[Duc97] Duchaineau, M et al. *ROAMing Terrain : real-Time Optimally Adapting Mashes*. Proceedings of Visualisation 1997. pp. 91-88.

[GMMN05] V. Gouranton1, S. Madougou, E. Melin and C. Nortet. Laboratoire d'Informatique Fondamentale d'Orléans, LIFO. *Interactive rendering of massive terrains on PC clusters*. EUROGRAPHICS - IEEE VGTC Symposium on Visualization (2005)

[DeBoer00] De Boer, W.H. *Fast Terrain Rendering using Geometrical MipMapping*. www.flipcode.com.

[LoHo04] Losasso, F. and Hoppe, H. *Geometry Clipmaps : Terrain Rendering Using Nested Regular Grids*. Proceedings of the 2004 SIGGRAPH Conference. pp. 769-776.

[GLR00] *OpenGL extension registry*. http://oss.sgi.com/projects/ogl-sample/registry/

[PHON75] B.T. Phong, *Illumination for Computer Generated Pictures*, Communications of the ACM, 18(6) :311-317, June 1975.

[ECS00] *Base de donnée de tables d'élévations de la NASA*. FTP://e0mss21u.ecs.nasa.gov

Projet IN55: visualisation de terrain
Diagramme de classe général
Par Béatrice Frey et Cyril Crassin, 11 Juin 2005
Classes propres à l'application
Classes de notre toolkit GLUtils
VertDataType
LODLevel
Texture
HeightMap
TerrainBlock
Vector2<T>
Vector3<T>
Vector4<T>
Frustum
CCamera
Camera
Renderable
Shader
QuadTree
TerrainGeoMipMap
ShaderCg
GLAppTerrain
PBuffer
GLApp
HeightMapPart
HeightMapPartIGT
HeightMapIGT
<use>
voisins
contient
fils