M i n i s t è r e d e l ' E d u c a t i o n N a t i o n a l e

UNIVERSITE MONTPELLIER II
UFR

IUP
GENIE MATHEMATIQUE ET INFORMATIQUE

RAPPORT DE STAGE

effectué pour

l'Institut de Recherche pour le Développement

à la Maison de la Télédétection

du 01/09/2006 au 31/12/2006

par

D U M I N I L J u l i e n

Directeur de stage :

M SOURIS Marc, Directeur de Recherche

Tuteur pédagogique :

Mme AHRONOVITZ Yolande

Quito3D

Développement d'une application interactive 3D
de visite virtuelle en milieu urbain destinée au grand public.

Remerciements

Je remercie l'Institut de Recherche pour le Développement, et plus particulièrement la Maison de la Télédétection, pour nous avoir chaleureusement accueillis durant ces quatre mois de stage.

Je remercie particulièrement Olivier Fouré, avec qui ce fut un plaisir de faire ce stage.

Merci à Marc Souris pour nous avoir proposé ce stage et nous avoir guidés.

Sans oublier, un grand merci à :

Monique Martiny et Chloé pour avoir répondu à tous nos besoins.

Yolande Ahronovitz pour ses conseils.

Florent Demoraes pour nous avoir consacré beaucoup de son temps.

Nicolas et tous les stagiaires de la MTD pour les bons moments passés ensemble.

Et merci à Célia, d'avoir été là dans les moments difficiles, comme dans les bons moments.

Sommaire

Introduction **3**
I. Etat de l'art **4**
1. Vue d'ensemble **4**
2. ROAM (*Real-time Optimally Adapting Meshes*) **5**
3. Continuous Level Of Detail (niveau de détail continu) **6**
II. Solution élaborée **7**
1. Principe de l'algorithme **7**
2. Deux méthodes **10**
3. Gestion d'une mosaïque de cartes **11**
III. Mise en oeuvre **12**
1. Technologies employées **12**
a) DirectX 12
b) Lua 12
2. Structures de données **13**
3. Interface utilisateurs **15**
Conclusion **16**
Glossaire **17**
Bibliographie **18**
Annexes **19**
Rapport technique **25**

Introduction

Dans le cadre d'une exposition sur la cartographie de la ville de Quito (Equateur) à travers les siècles en décembre 2006 et janvier 2007, il a été demandé à l'Institut de Recherche pour le Développement (IRD) de réaliser une application interactive performante de visite virtuelle sur la ville pour le grand public.

L'Institut de Recherche pour le Développement est un établissement public à caractère scientifique et technologique, qui conduit des programmes scientifiques centrés sur les relations entre l'homme et son environnement dans les pays du Sud, dans l'objectif de contribuer à leur développement. Ses missions fondamentales sont la Recherche, l'Expertise et la Valorisation, le Soutien et la Formation, et l'Information scientifique.

C'est au sein de la Maison de la Télédétection de Montpellier que nous avons été accueillis. Elle regroupe des équipes de recherche de différents centres et instituts de recherche (dont l'IRD) pour constituer un pôle de recherche appliquée en télédétection et information géographique.

Les solutions existantes de visites virtuelles à l'heure actuelle ne sont soit pas adaptées au volume de données, soit pas assez orientées visite virtuelle.
Nous avons donc été chargés, avec Olivier Fouré, de concevoir et développer un logiciel de survol 3D de la région métropolitaine de Quito, à partir du modèle numérique et de photographies aériennes à grande échelle.

Le logiciel propose à l'utilisateur plusieurs modes de déplacement, dont un simulateur de vol et une liste de vols prédéfinis.
Le caractère grand public impose une simplicité d'utilisation et la fiabilité du logiciel.

Le développement s'est fait en C++ (avec Visual Studio 2005) et utilise la librairie DirectX 9. Pour conserver une bonne qualité d'image et assurer la fluidité nécessaire à la simulation, nous avons dû intégrer les techniques les plus performantes en visualisation de terrain 3D.

L'application est optimisée pour les machines multiprocesseurs, mais tourne sans problème sur une machine monoprocesseur. Pour des raisons matérielles, le logiciel a été limité dans le rendu temps réel, et il ne pourra montrer tout son potentiel que sur les futurs ordinateurs.

La surface couverte, comprenant Quito et ses environs, est d'environ 20 km sur 44 km, avec une précision d'image de 1 m. Le logiciel peut également facilement être adapté à d'autres régions.
Il reste malgré cela un grand potentiel d'amélioration, comme ajouter le rendu des bâtiments ou suivre le cheminement des rues. Par la suite, le logiciel sera peut-être amené à des utilisations plus ludiques, comme un simulateur de vol plus évolué, ou un simulateur de courses de voitures.

I. Etat de l'art

1. Vue d'ensemble

A l'ère de la 3D, il existe de nombreuses techniques d'optimisation pour le rendu 3D en temps-réel. Ces techniques se divisent en trois grandes catégories. La première est le problème de visibilité, qui consiste simplement à décider si un objet est considéré comme visible ou non (par exemple, un objet derrière la caméra n'est pas visible). La deuxième catégorie d'optimisations est le choix du niveau de détail (en anglais, *Level Of Detail*), qui donnera un rendu plus grossier pour un objet lointain que pour le même objet plus proche. Et enfin la troisième réside dans l'optimisation du code source, en utilisant les solutions proposées par les cartes graphiques et en apportant un soin particulier aux fonctions appelées plusieurs milliers de fois par image.

Le rendu de terrains est pourvu lui aussi de son lot d'optimisations. Alors que les problèmes de visibilité et d'optimisation du code ne seront pas différents de la plupart des applications 3D, la gestion du niveau de détail se révèle bien plus complexe. En effet, un terrain peut se trouver à la fois proche et loin de la caméra. De plus, une plaine demandera moins de précision qu'une montagne. On parlera plus précisément de simplification du relief. Il existe de nombreux algorithmes dans ce but, dont on peut trouver une liste assez exhaustive sur Virtual Terrain Project (http://www.vterrain.org/LOD/Papers/).

Notre premier critère de sélection s'est porté sur le format des données. En effet, pour le relief nous avons en notre possession une grille d'altitudes, nous ne nous sommes donc pas intéressés aux réseaux de triangles irréguliers. Ensuite, nous avions comme impératif de gérer le « *popping* », qui a pour effet de voir le relief changer brutalement quand on s'en approche. Nous avons donc laissé de côté les algorithmes, dits « *Chunked Level Of Detail* », qui consistent simplement en un découpage du modèle en plus petits morceaux, chargés en mémoire à un niveau plus ou moins précis (ci-dessous) selon certains critères.

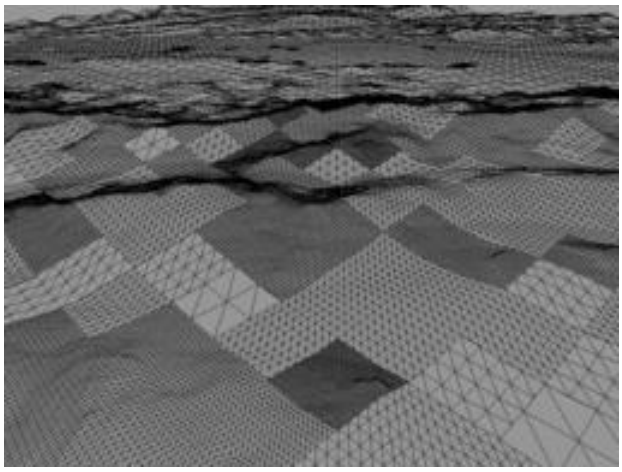


Figure 1 - Chunked Level Of Detail

Au final, deux algorithmes nous ont particulièrement intéressés : le ROAM (*Real-time Optimally Adapting Meshes*) et le *Continuous Level Of Detail* (niveau de détail continu).

Ces deux algorithmes ont la particularité intéressante de partir du modèle peu détaillé vers un modèle plus détaillé, ce qui permet de ne traiter qu'une partie des données à la fois. De plus, ils offrent un rendu dynamique du relief, conservant un bon réalisme sans perte de performances.

2. ROAM (*Real-time Optimally Adapting Meshes*)

Le ≪ *ROAMing* ≫ est un algorithme de simplification de terrain créé par Mark Duchaineau. Il a été par la suite adapté à la simplification d'autres objets virtuels que des terrains. Son principe est basé sur un arbre binaire de triangles, donc chaque triangle peut être divisé en deux triangles fils ou fusionné avec un triangle voisin pour reformer le triangle parent. Il introduit la notion de structure en diamant, que nous reverrons plus tard, et définit clairement les répercutions de la division d'un triangle sur les triangles voisins.

Les avantages de cet algorithme sont la flexibilité du critère de division/fusion, une qualité optimale du relief pour un nombre de triangles donné, la possibilité de choisir un nombre d'images par secondes strict, et la cohérence d'une image à l'autre. Son point fort est l'utilisation de deux files de priorité afin de définir quel est le prochain triangle à diviser (respectivement fusionner). C'est également son point faible car maintenir une file de priorité a un coût important sur un grand volume de données.

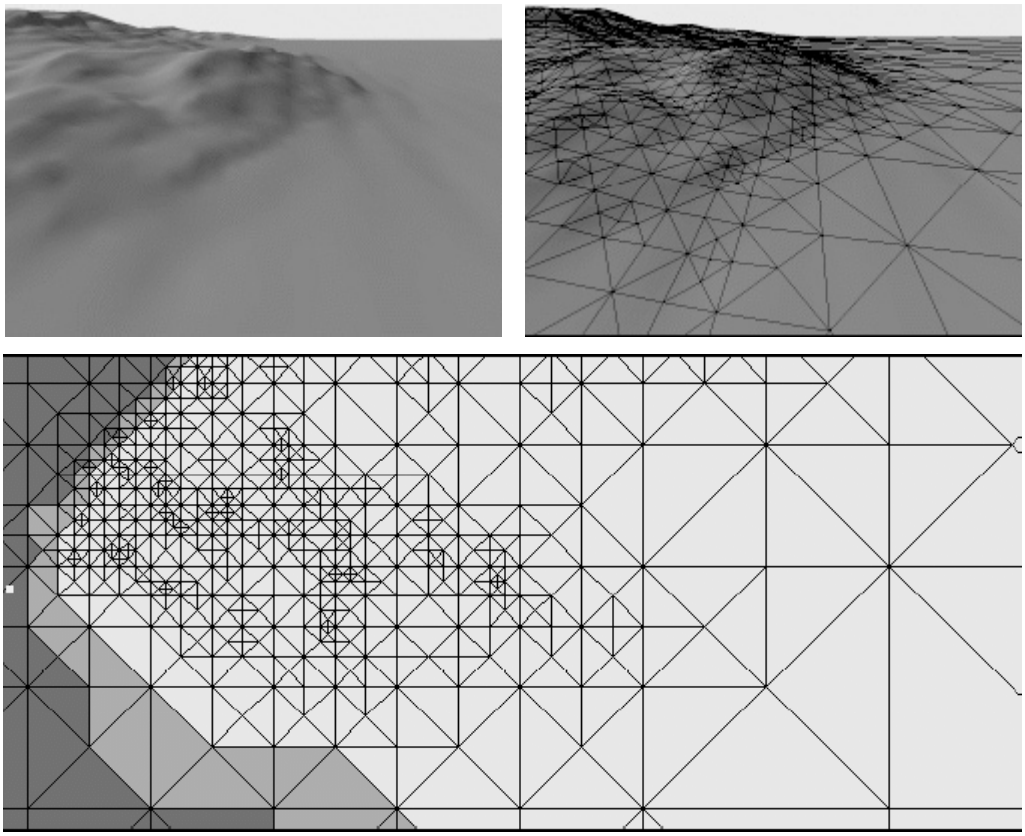


Figure 2 - ROAM en mode texturé, fil de fer et vue de dessus

3. Continuous Level Of Detail (niveau de détail continu)

Le ≪ *Continuous LOD* ≫ est un algorithme créé par Peter Lindstrom. Il est basé exclusivement sur des grilles régulières d'altitudes (de côté une puissance de 2 plus 1) et utilise un arbre quaternaire de points (chaque point a quatre fils et deux parents). Il a introduit la notion de taux d'erreur, ou écart, qui consiste à mesurer la plus grande différence d'altitude possible dans un bloc de triangles.

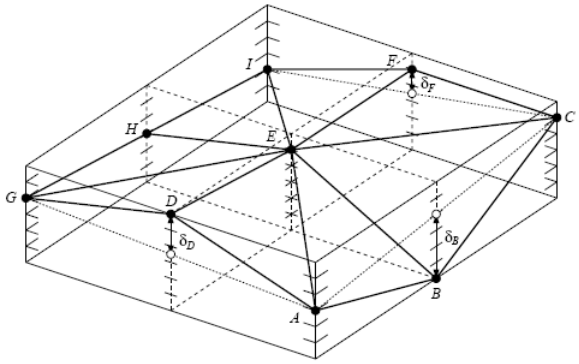


Figure 3 - Notion d'écart

Les avantages de cet algorithme sont sa rapidité, sa réduction très faible de la qualité, la génération de niveaux de détail en temps-réel, et le paramétrage du taux d'erreur maximal, le tout à un nombre d'images par secondes constant. Son point faible est l'utilisation d'un arbre quaternaire de points, qui rend plus difficile sa compréhension et son implémentation.

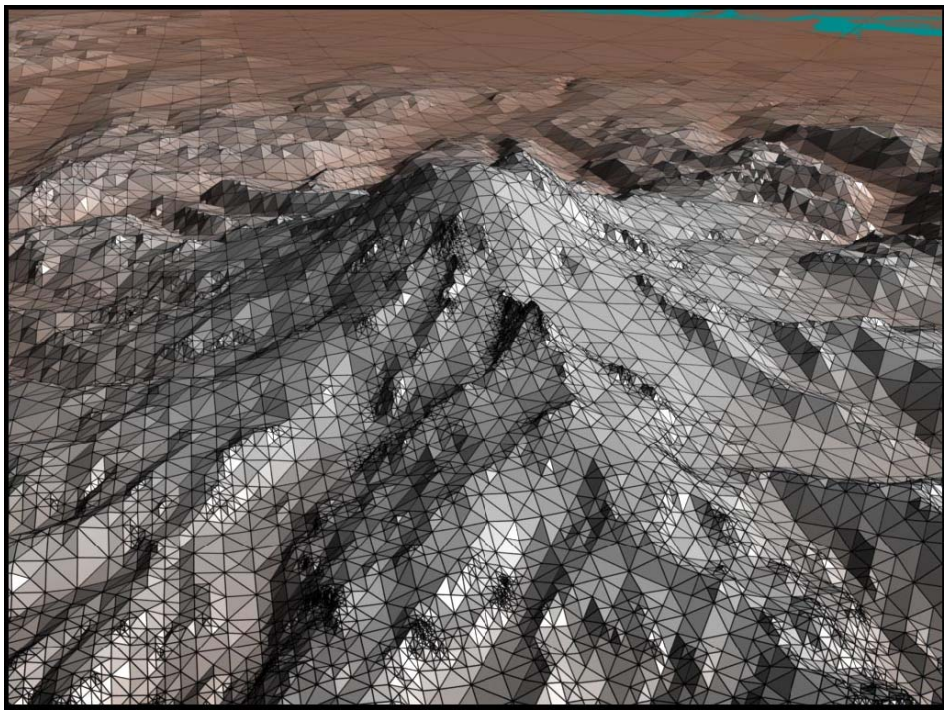


Figure 4 - Continuous Level Of Detail

II. Solution élaborée

1. Principe de l'algorithme

Des deux algorithmes cités avant, nous avons créé le notre alliant un maximum d'avantages des deux, basé sur un arbre binaire de triangles. En pratique, nous partons de deux triangles de base, qui peuvent chacun être récursivement divisés en deux triangles fils. Comme nous n'avons pas un mais deux triangles de base (pour former un carré), nous les considérons comme des descendants d'un ancêtre commun (Figure 5). Nous utilisons la numérotation du parcours en largeur d'un arbre binaire complet.

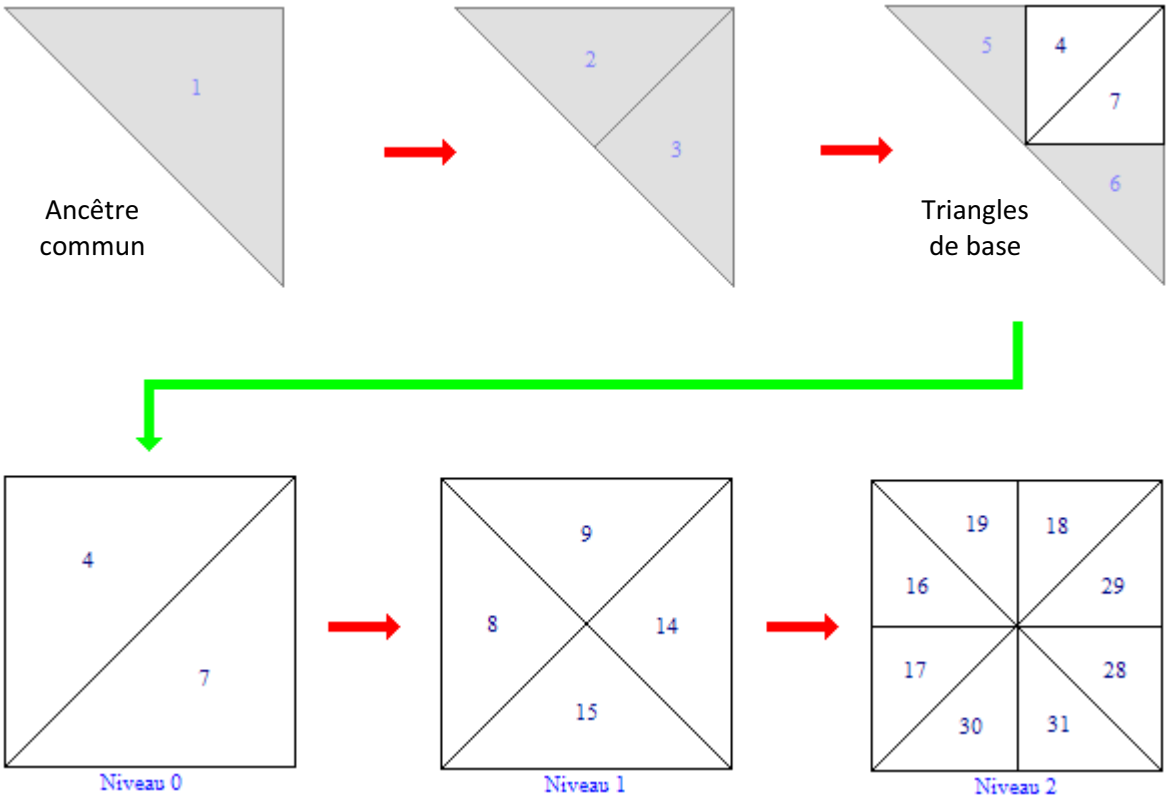


Figure 5 - Triangles de base, ancêtre commun et premiers niveaux de division

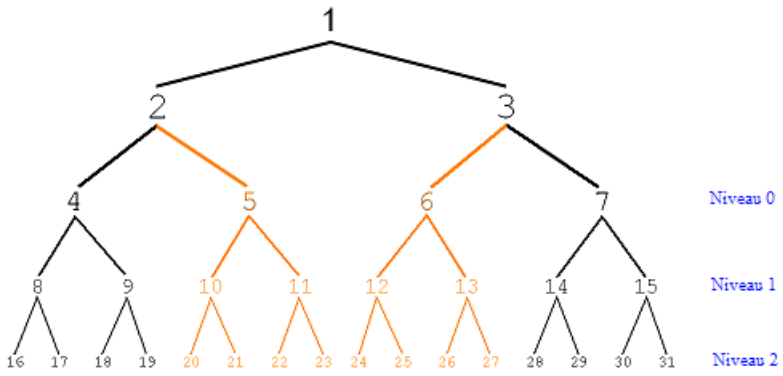


Figure 6 - Arborescence des premiers niveaux de division

Sur chacun de ces triangles, il peut être effectué deux types d'opérations : une division ou une fusion.

La division permet d'affiner le modèle. Elle consiste à couper un triangle parent selon l'axe passant par son angle droit et le milieu de son hypoténuse. On obtient ainsi deux triangles fils, eux-mêmes rectangles.

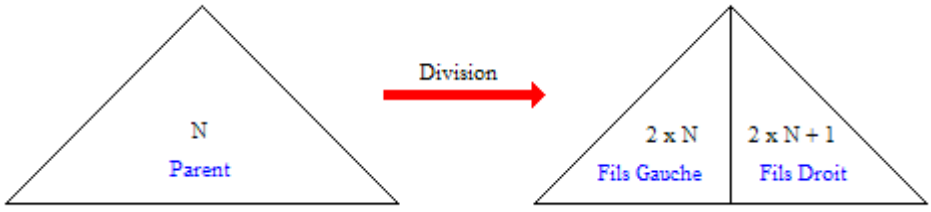
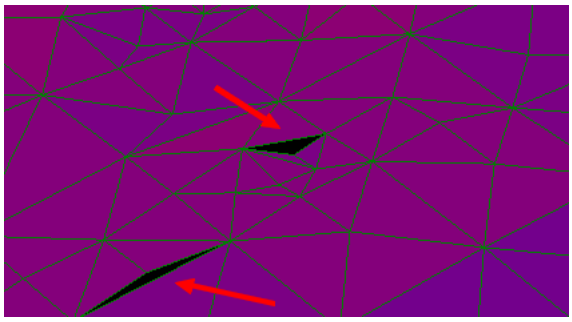


Figure 7 - Division d'un triangle

La difficulté vient du fait qu'en faisant ainsi, on peut créer des trous, aussi appelé « *cracking* ». En voici un exemple :



La division d'un triangle sans diviser son voisin d'hypoténuse entraîne systématiquement l'apparition d'un trou. Cependant, son voisin d'hypoténuse n'est pas forcément au même niveau de division, auquel cas il faut le diviser récursivement. Son voisin d'hypoténuse est enfin de même niveau et peut être lui aussi divisé (cf. Annexe I). Ainsi, deux triangles voisins ont au maximum un niveau de différence.

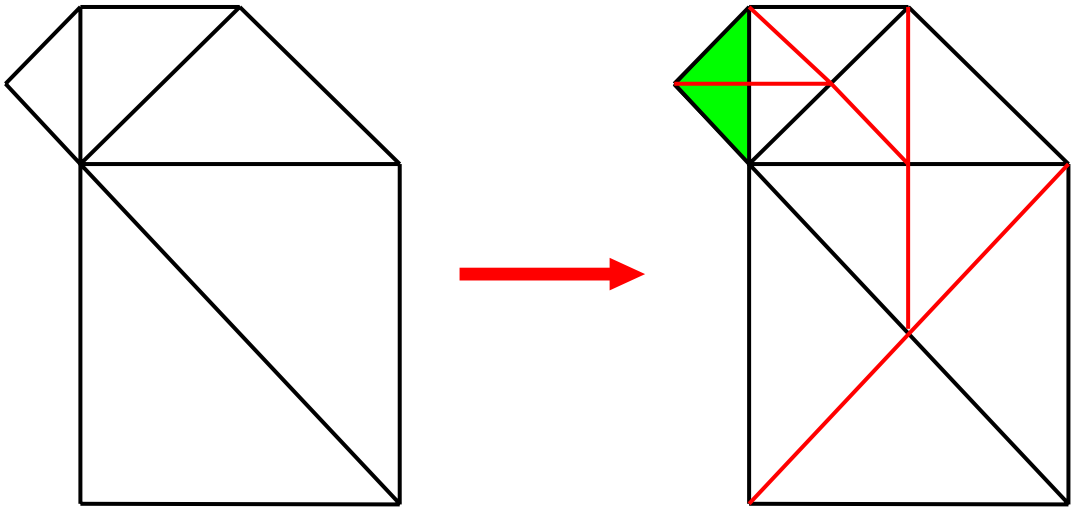


Figure 8 - Exemple de division en série pour éviter le *cracking*

La fusion, quant à elle, permet de simplifier le relief. Elle est plus simple car elle n'a pas le caractère récursif. Elle doit seulement remplir certaines conditions. La première est que les deux frères (entendez par là les deux fils du parent) doivent être marqués comme à afficher. La deuxième

est que si le parent a un voisin d'hypoténuse (s'il n'est pas sur le bord de la carte), alors les deux fils de ce voisin (appelés cousins) doivent eux aussi être marqués comme lus.

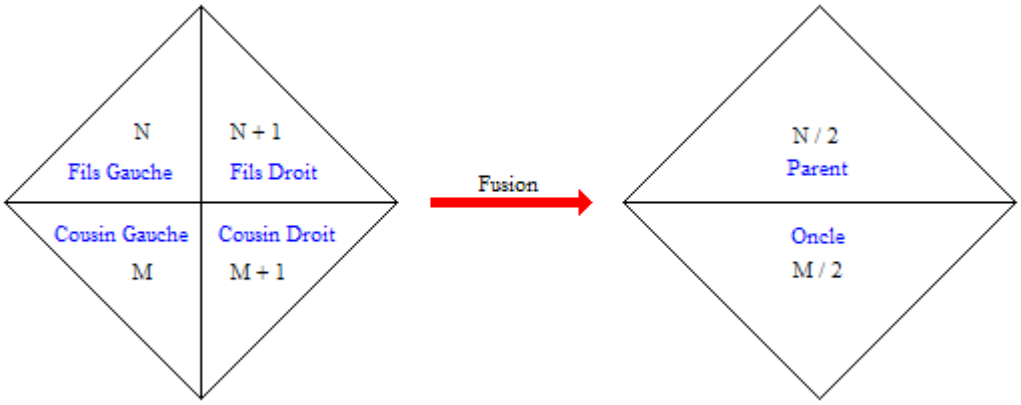


Figure 9 - Fusion de quatre triangles en deux triangles parents

La disposition des triangles de la figure ci-dessus après la fusion est appelée un diamant. Elle a la particularité de pouvoir être divisée sans récursivité.

Les quatre coordonnées des deux triangles de base (Figure 10) sont les coordonnées des coins de la grille d'altitude. Pour s'assurer que le nouveau point créé par une division est toujours aligné sur la grille, on travaille avec une grille d'altitude de coté $2^n + 1$. Le niveau de division maximum d'un triangle est alors $2 * \log_2(n-1) - 2$. Par exemple, un triangle de base d'une grille de taille 5x5 doit être divisée 4 fois avant d'être au niveau de division maximum. De même, dans cette même grille, un triangle de niveau 4 ne pourra plus être divisé, et recouvrira une demi case de la grille (une case correspond à un carré formé par quatre points adjacents de la grille).

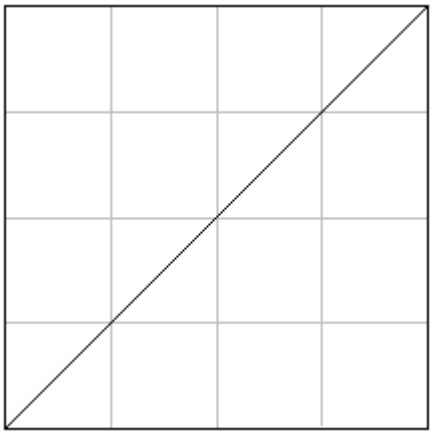


Figure 10 – Triangles de base sur une grille 5x5

2. Deux méthodes

➢ Première méthode

Nous avons d'abord mis en place un premier algorithme qui, pour chaque image, parcourt l'arborescence en profondeur depuis les deux triangles de base jusqu'au niveau désiré. Lors du parcours, pour chaque triangle, on décide si le triangle doit être ou non divisé. Si c'est le cas, on continue le parcours des deux fils, sinon on marque le triangle comme à afficher. Il ne faut pas oublier de marquer les triangles comme à afficher lors de la division récursive.

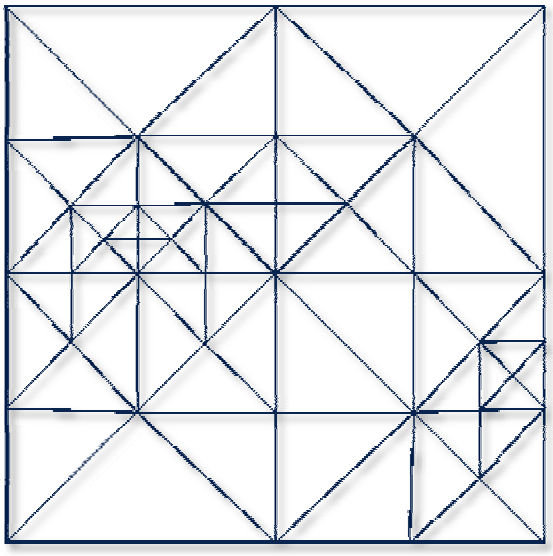


Figure 11 - Exemple de niveau de détail sans *cracking*

Cet algorithme a l'avantage d'être simple à mettre en place, nous permettant de tester rapidement notre travail. Son avantage est que la fusion n'a pas lieu d'être utilisée. Mais il montre vite ses limites quand nous augmentons le volume des données car à chaque image, l'arborescence est parcourue et la totalité des triangles sont envoyés à la carte graphique.

➢ Seconde méthode

Une fois ce premier algorithme bien rodé, nous avons implémenté un algorithme beaucoup plus difficile à mettre en place, mais bien plus performant sur de grands volumes de données. Cet algorithme consiste à conserver l'arborescence courante pour l'image suivante, en stockant les points et les triangles dans la mémoire de la carte graphique, et en ne modifiant que ceux qui changent d'une image à l'autre.

La difficulté vient du fait que nous devons alors maintenir deux tableaux sans trous dans la carte graphique : le tableau des points utilisés, et le tableau des triangles utilisés (indices des points qui forment les triangles). Pour la division, il suffit de rajouter les points et triangles créés à la suite du tableau, mais pour la fusion, il faut déplacer les derniers éléments des tableaux dans les trous créés et mettre à jour tous les éléments concernés par ces déplacements. Pour ce faire, il faut stocker un certain nombre d'informations supplémentaires, comme pour chaque triangle la liste de ses voisins, pour chaque point la liste des triangles qui l'utilisent, etc.

3. Gestion d'une mosaïque de cartes

Les cartes graphiques actuelles gèrent difficilement les textures dépassant 2048x2048 pixels. Nous avons donc dû couper notre carte en plusieurs petites cartes afin que les morceaux de photo satellite correspondants ne dépassent pas les 2048x2048 pixels. De plus, nous ne pouvons pas nous permettre de mettre la texture de 2048 de côté sur toutes les cartes. Nous avons donc créé pour chaque carte plusieurs niveaux de détail de la texture.

Avec ce système, on peut très bien imaginer que deux cartes côte à côte n'ont pas exactement le même relief sur leur bord commun (produisant du *cracking*). Pour pallier à ce problème, nous avons employé une technique très simple dite de la mini-jupe (en anglais, *mini-skirt*). Elle consiste à rajouter des triangles qui partent du bord de la carte et descendent un peu plus bas, un peu comme une nappe qui pend sur les bords d'une table.

Afin de gagner en vitesse de calcul, on peut également détecter si une petite carte est, ou non, dans le champ de vision. Cette technique, appelée « *frustum culling* », consiste à calculer les six plans qui composent le champ de vision, et de déterminer ensuite si la petite carte est derrière l'un de ces plans, auquel cas elle n'est pas visible.

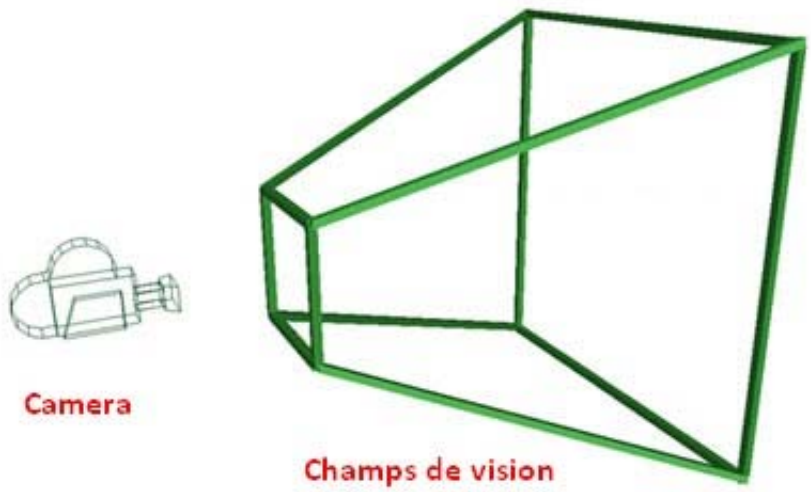


Figure 12 - Camera et champ de vision

III. Mise en oeuvre

1. Technologies employées

a) DirectX

DirectX est une librairie C/C++ (entre autres) développée par Microsoft permettant de développer des applications 3D et de tirer parti de la carte graphique. Mais elle n'est pas limitée à la 3D, car elle permet entre autre de charger des images, du son ou même de gérer les périphériques d'entrée comme le clavier et la souris.

Pour ce qui est de l'affichage, DirectX prend en entrée soit un tableau de vertices (qui comprend dans notre cas la coordonnée 3D d'un point, ainsi que la coordonnée 2D de la texture) et un tableau de triangles (chacun composé des indices des trois vertices le composant), soit directement un tableau de triangles (qui dans ce cas sont chacun directement composés de trois vertices). La deuxième solution n'est pas optimale car deux triangles voisins sont composés d'un même vertice, qui se retrouve alors dupliqué.

Enfin, cette librairie comprend également un nombre important de fonctions de calcul 3D (opérations sur les matrices, produit vectoriel, …), de projection, d'intersections, etc. Son seul gros inconvénient est qu'elle n'est pas portable et ne fonctionne que sous Windows (contrairement à OpenGL, son principal concurrent). Cependant, il reste envisageable de porter notre application sous OpenGL, même si celui-ci ne s'occupe que de la 3D.

b) Lua

Lua est un langage de script simple, léger, rapide et portable. Il est de plus en plus utilisé dans les jeux et autres applications. Il est souvent utilisé pour gérer le comportement des applications (mouvement de la caméra, intelligence artificielle, …) ou laisser la possibilité à l'utilisateur de rajouter/personnaliser des fonctionnalités sans avoir à recompiler le logiciel.

Dans notre cas, nous utilisons Lua pour gérer la caméra en mode chemin prédéfini, pour rajouter des chemins prédéfinis et pour rajouter des noms sur des lieux. Nous y reviendrons un peu plus loin.

2. Structures de données

Nous allons étudier les structures de données utilisées pour chaque petite carte.

- Données stockées dans la mémoire vive du GPU (mémoire de la carte graphique).

```
struct GPUVertice
{
	D3DXVECTOR3 position;
	float u, v;
};
```

Nous avons d'abord un tableau de vertices, qui contiennent chacun la position d'un point dans l'espace, et les coordonnées projetées de ce point sur la texture de la carte. Des coordonnées de texture s'expriment dans l'intervalle [0 ; 1]. Seuls les points utilisés par au moins un triangle y sont stockés.

```
struct GPUTriangle
{
	int a, b, c;
};
```

Puis nous avons un tableau de triangles, qui contiennent chacun les indices des trois vertices dans le tableau de vertices. Par exemple, si *t* est un *GPUTriangle*, et *vertices* le tableau de vertices, alors *vertices[t.a]* est un *GPUVertice*.

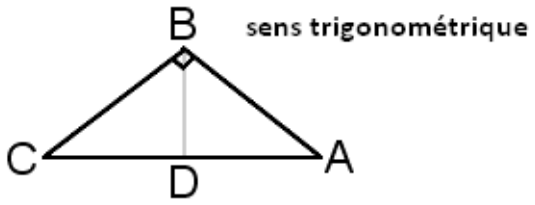


Figure 13 - Convention de nommage des points d'un triangle

Pour un gain de mémoire et de performance, nous ne créons les tableaux de vertices et de triangles de taille environ égale à 10% du modèle complet.

- Données stockées dans la mémoire vive du CPU (mémoire de l'ordinateur).

Pour accélérer les accès à la mémoire de la carte graphique, celle-ci est utilisée en lecture seule. C'est pourquoi on stocke les mêmes informations en double dans la mémoire vive de l'ordinateur. De plus, nous y stockons d'autres informations nécessaires à notre algorithme.

```
struct CPUPoint
{
	int vertice;
	GPUVertice gpuvertice;
	float ecart;
};
```

Pour chaque point de la grille d'altitude (qu'il fasse parti d'un triangle ou non), nous stockons l'indice du vertice correspondant (ou -1 si aucun), le *GPUVertice* précalculé (afin d'éviter de le recalculer à chaque fois qu'on rajoute un vertice, et l'écart. L'écart est la valeur maximale entre l'altitude du point concerné et l'altitude de la moyenne des points A et C d'un triangle dont ce point en est le point D. Il correspond plus simplement à la différence entre l'altitude réelle de D et l'altitude du projeté de D sur [AC]. Pour plus de précision, on tient également compte des autres points qui sont sur [AC].

Ensuite, nous avons les tableaux de vertices et de triangles qui sont de même taille que dans la mémoire du GPU et qui ont leurs éléments indexés dans le même ordre (à même indicé, même élément).

```
struct CPUVertice
{
	int point;
	int triangle;
};
```

Pour un vertice, nous avons besoin de connaître l'indice du point correspondant (avec lequel nous connaissons le *GPUVertice*), et l'indice d'un des triangles qui utilise ce vertice. Nous pouvons retrouver les autres triangles qui l'utilisent en tournant autour du point grâce aux voisins de ces triangles.

```
struct CPUTriangle
{
	int id;
	char niveau;
	int vg, vd, vb;
	GPUTriangle gputriangle;
};
```

Enfin, pour un triangle, nous avons besoin de connaître son identifiant pour savoir si c'est un fils gauche ou droit (pair ou impair). Nous stockons son niveau de division (qui aurait pu être calculé à partir de l'identifiant, mais cela aurait demandé plus de calculs), afin de savoir si l'on est ou non au premier ou dernier niveau (et donc si l'on peut le diviser/fusionner). Nous stockons également les indices de ses trois voisins (gauche, droite et base/hypoténuse), partie la plus difficile à maintenir à jour. Et sans oublier une copie du *GPUTriangle*, afin de connaître les indices des vertices qui le compose.

Avec toutes ces structures, nous arrivons donc à maintenir des tableaux de vertices et de triangles sans trous.

3. Interface utilisateurs

L'utilisateur peut choisir le mode de déplacement dans une liste déroulante. Il a alors le choix entre le mode libre, où il peut se déplacer sur la carte à son gré, le mode avion, où la caméra avance en continu à vitesse variable, le mode à ras du sol, où la caméra suit le relief, et le mode chemins prédéfinis, où il peut choisir une visite virtuelle prédéfinie parmi une autre liste déroulante.

Dans le mode avion, nous avons empêché toute collision en redressant le nez de l'appareil dès qu'il est à trop basse altitude. De même, pour éviter que l'utilisateur se retrouve coincé dans un coin de la carte, l'avion amorce un virage s'il s'approche trop d'un des bords.

Dans le mode à ras du sol, l'utilisateur peut, outre le déplacement, faire varier l'inclinaison avec la molette de la souris, et avec, faire varier légèrement l'altitude de la caméra. Mais lors des déplacements, l'altitude par rapport au sol reste constante.

Pour les chemins prédéfinis, nous avons utilisé le langage de script Lua. A partir de la position et des angles de rotation donnés à un moment t et $t + 1$, nous faisons une simple interpolation linéaire de la position et des angles.

L'interface affiche également en temps-réel l'altitude réelle de la caméra. Le nom de certains lieux est affiché juste au dessus de leur position. Nous avons également ajouté du vent comme bruit d'ambiance sonore.

Conclusion

Après plusieurs mois de travail, nous avons été heureux de voir le résultat obtenu. Malgré quelques lenteurs qui nous ont obligés de réduire légèrement la qualité, le rendu est fluide et agréable. Après des difficultés de mise en place en Equateur (où nous aurions aimé assister) pour des raisons encore inconnues, le logiciel s'est ensuite avéré être très stable et avoir fortement satisfait les organisateurs. Cependant, le temps ne nous a pas permis certaines améliorations, comme l'ajout des bâtiments en relief ou encore la recherche du plus court chemin dans les rues.

Et même si le décalage horaire (+6h chez notre tuteur en Thaïlande, -6h au lieu d'exposition en Equateur) et le fait que les organisateurs de l'exposition ne parlent pas français ont rendu la communication plus difficile, nous sommes restés en contact régulier jusqu'à l'inauguration de l'exposition. Nous aurions cependant aimé avoir plus d'échos sur les impressions des visiteurs.

Ce stage m'a permis de connaître une expérience dans un domaine que j'apprécie beaucoup, qui est le développement d'applications en trois dimensions, au sein d'une entreprise. Il m'a donné plus que jamais envie de continuer dans cette voie, et m'a également montré combien il est parfois difficile de respecter des délais. Maintenant, nous espérons avoir l'occasion de continuer à travailler sur ce moteur de visites virtuelles.

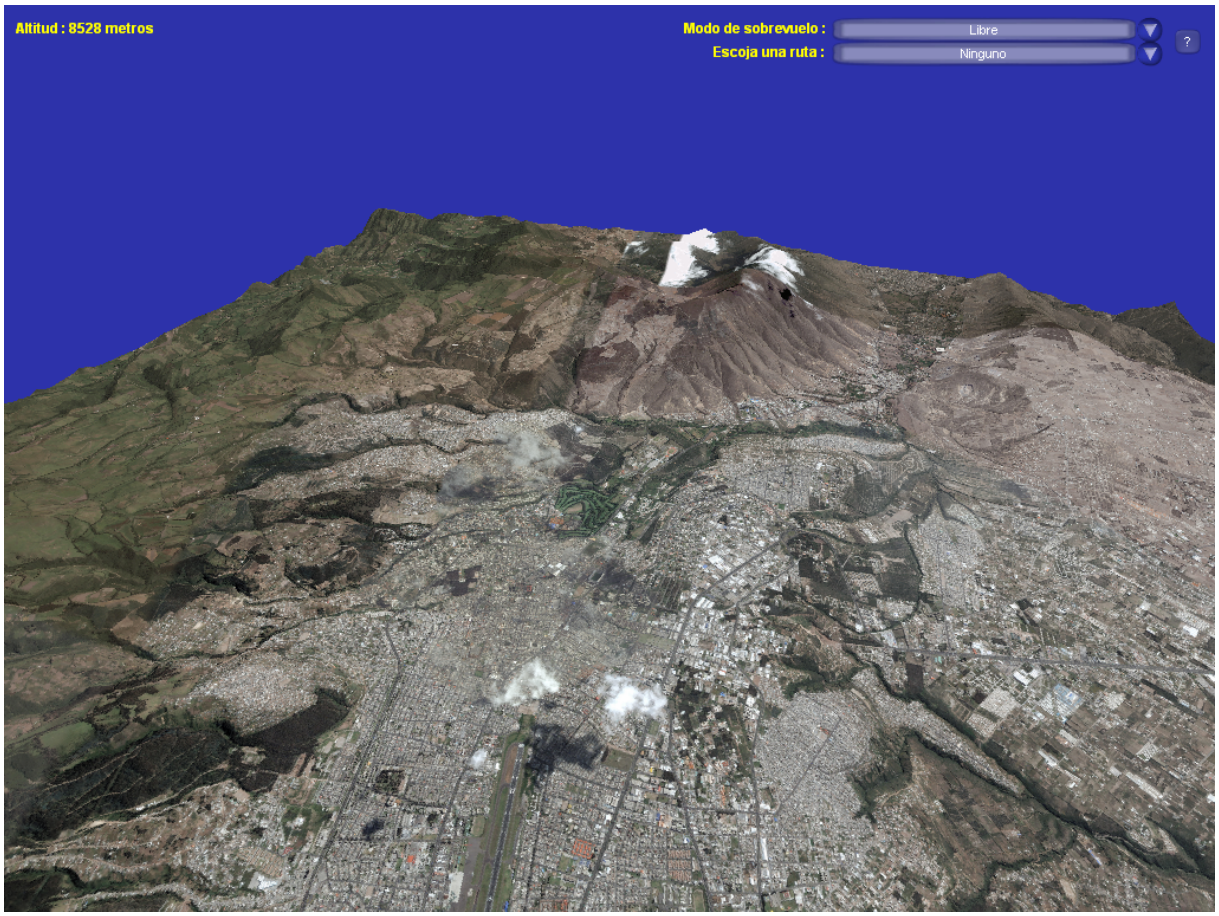


Figure 14 - L'application dans toute sa splendeur

Glossaire

Rendu 3D : calcul et affichage d'une scène virtuelle en trois dimensions sur l'écran.

Temps-réel : en opposition à précalculé, qui opère le calcul à chaque sollicitation.

Modèle numérique : données permettant de représenter un objet ; dans le cas d'un terrain, il s'agit souvent d'une grille d'élévations (en anglais, *heightmap*).

MNT : Modèle Numérique de Terrain (cf. Modèle numérique).

Texture : image pouvant être plaquée sur un objet virtuel.

GPU : *Graphic Processor Unit*, le processeur de la carte graphique.

CPU : *Central Processing Unit*, le processeur de l'ordinateur.

Bibliographie

Virtual Terrain Project : Liste de documents traitants du niveau de détail des terrains.
http://www.vterrain.org/LOD/Papers/

Real-Time, Continuous Level of Detail Rendering of Height Fields par Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust.
http://www.cc.gatech.edu/gvu/people/peter.lindstrom/papers/siggraph96/

Visualization of Large Terrains Made Easy par Peter Lindstrom, Valerio Pascucci.
http://www.gvu.gatech.edu/people/peter.lindstrom/papers/visualization2001a/

ROAMing Terrain: Real-time Optimally Adapting Meshes par Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, Mark B. Mineev-Weinstein.
http://www.cognigraph.com/ROAM_homepage/index.html

DirectX 9.0 Programmer's Reference par Microsoft, fourni avec le kit de développement de DirectX.

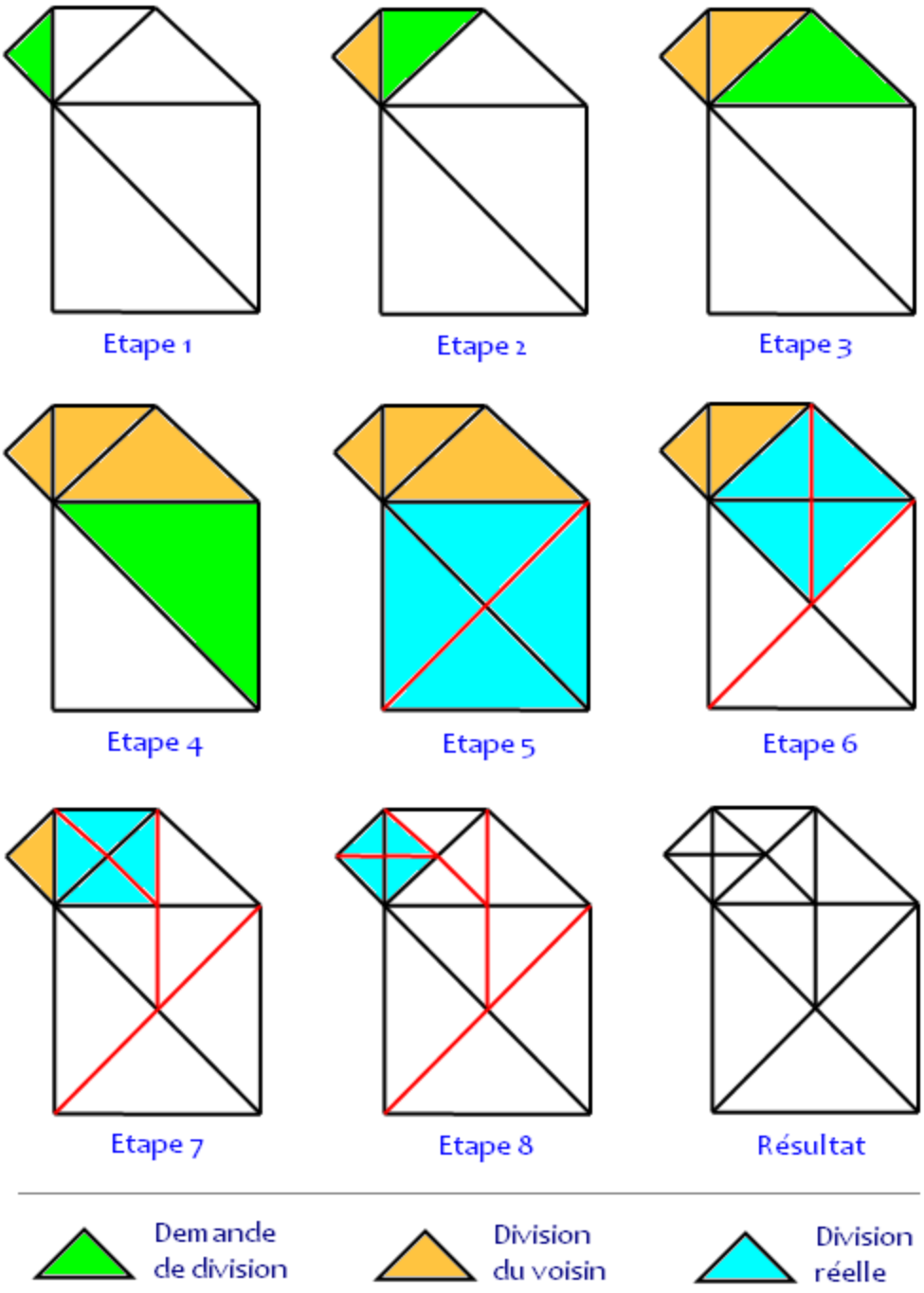
DirectX 9, Programmation de jeux 3D par Laurent Testud, édité par CampusPress.

Maison de la télédétection (MTD) en Languedoc-Roussillon.
http://www.teledetection.fr/

Institut de Recherche pour le Développement (IRD).
http://www.mpl.ird.fr/

Annexes

I. Division récursive d'un triangle sans *cracking*

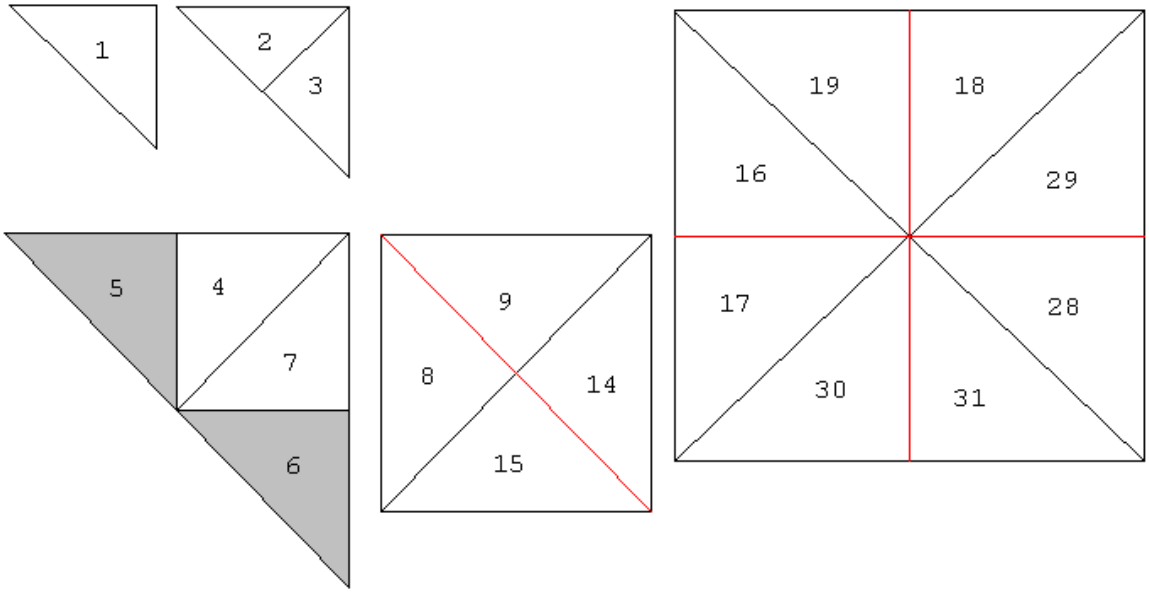


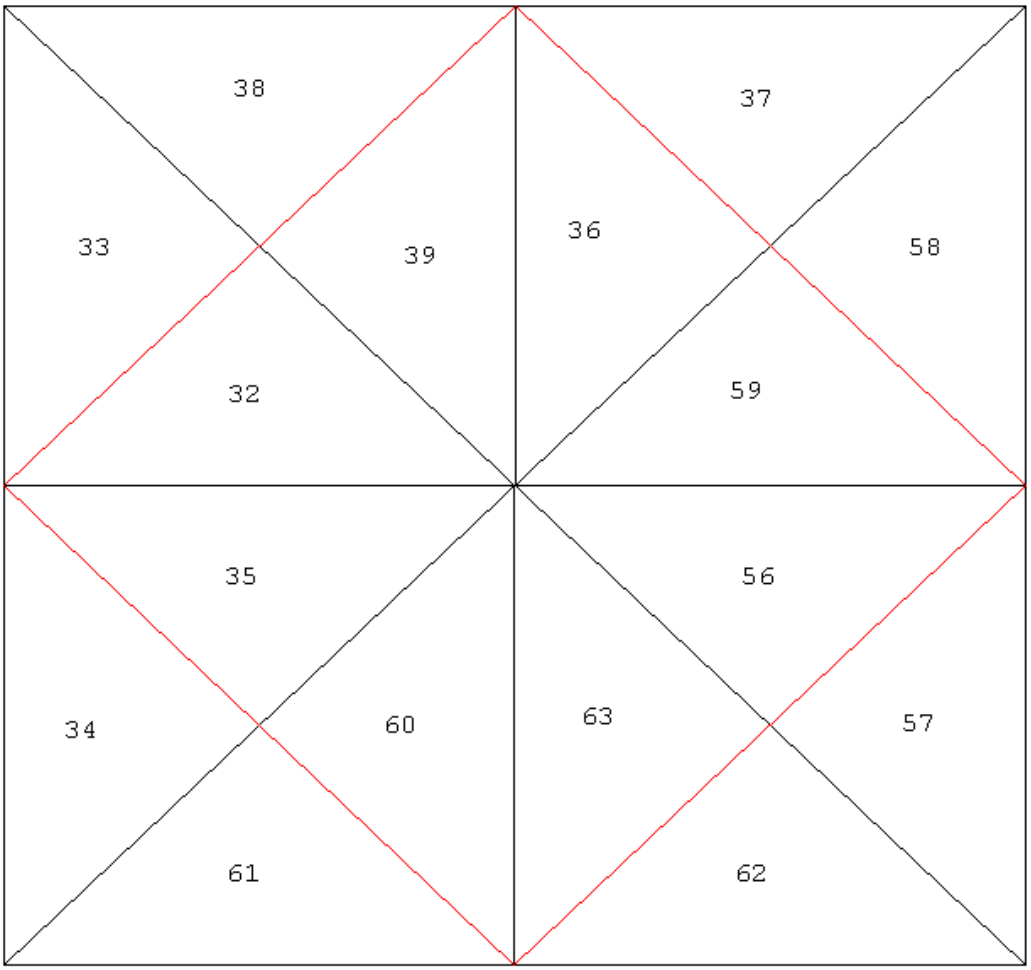
II. Quadrillage pour faire des traces de l'algorithme sur une grille 9x9

On peut facilement y reproduire un niveau de détail particulier, y noter les identifiants de triangles et des points, ainsi que marquer les points.

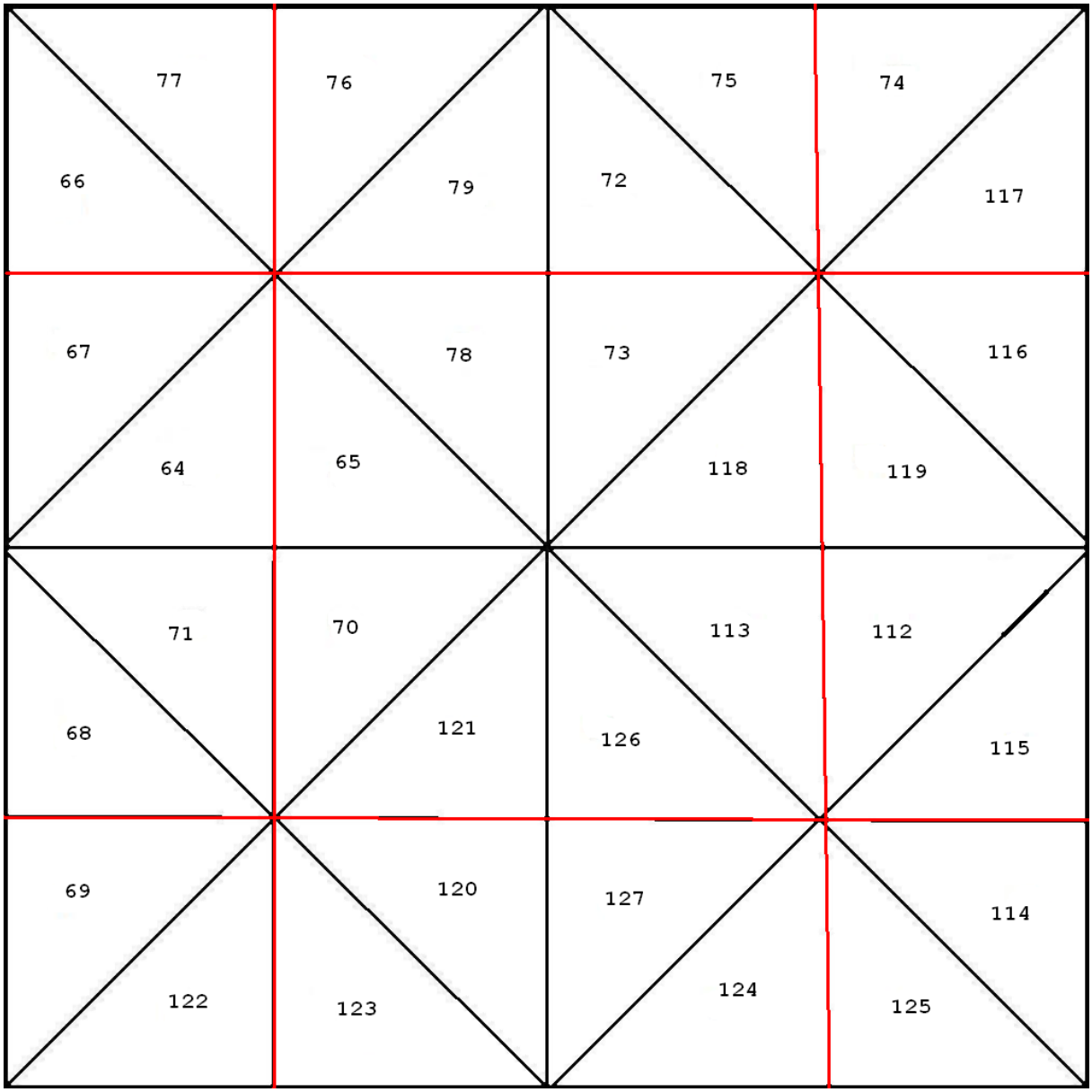
III. Numérotation des triangles jusqu'au niveau 4

➢ **Niveaux 0 à 3**





➢ **Niveau 4**


77
76
75
74
66
79
72
117
67
78
73
116
64
65
118
119
71
70
113
112
68
121
126
115
69
120
127
114
122
123
124
125

IV. Description des commandes de navigation en fonction du mode de survol choisi

1 Mode libre :

Avancer / Reculer 

Tourner à gauche / à droite  ou 

Incliner en avant / en arrière 

Décaler horizontalement vers la gauche / la droite 

Décaler verticalement vers le haut / le bas 

Monter / descendre 

2 Modo avion :

Tourner à gauche / à droite  ou 

Incliner en avant / en arrière 

Augmenter / réduire la vitesse 

3 Mode à ras du sol :

Avancer / Reculer 

Tourner à gauche / à droite  ou 

Décaler horizontalement vers la gauche / la droite 

Monter / descendre 

Rapport technique

- Configuration.

La configuration du logiciel s'effectue grâce au fichier « config.h ». On peux y changer les caractéristiques des données à charger :
Le thème de la surface, dans le cadre de ce stage « Quito ».
Les dimensions de la mosaïque de cartes.
L'extension des données, mas ne change pas la façon dont elles sont lues.
Les coordonnées du point de référence de la zone modélisé, en coordonnées projetées.
La précision et la résolution maximale des morceaux de relief et d'images satellites.
Le facteur d'accentuation du relief.
La résolution du mode plein écran.
Les touches de déplacement et les textes qui sont affichés.
Toute modification de ce fichier nécessite une recompilation.

- Format des données.

Les données doivent remplir certaine conditions, chaque fichier doit être de la forme :
Les morceaux de relief sont nommés « Data\Modeles\%1_%2_%3_%4.bin » avec :
%1 = Le thème de la surface modélisé
%2 = résolution de cette partie du relief égale à 2^n + 1 avec n >= 1
%3 = coordonnées de ligne dans la mosaïque de cartes, nombre positif
%4 = coordonnées de colonne dans cette même mosaïque, nombre positif
Le fichier bin contient %2 * %2 valeurs de type **float (4 octets)** au format binaire

Les images satellites sont nommées « Data\Textures\%1_%2_%3_%4.bmp » avec :
%1 = idem que le précédent
%2 = résolution de cette partie de l'image égale à 2^n + 1 avec n >= 1
%3 = idem que le précédent
%4 = idem que le précédent

- Ajouter un nom de lieu

Dans l'application, placez-vous au dessus du lieu concerné, et appuyez sur la touche L.
La ligne de script Lua est directement copiée dans le presse-papier.
Ouvrez ensuite le fichier *Data\Scripts\Locations.lua* et collez le texte à la suite des autres lieux.
Mettez le nom de votre choix, et sauvegardez le fichier.
Il faut relancer l'application pour qu'elle prenne en compte le nouveau lieu.

- Ajouter un chemin prédéfini

Ouvrez le fichier *Data\Scripts\Paths.lua*.
Ajouter un nouveau nom de chemin selon le même modèle que les autres.
Dans l'application, placez-vous à une position clé, et appuyez sur la touche P.
La ligne de script Lua est directement copiée dans le presse-papier.
Collez le texte à l'endroit voulu, et donnez-lui un numéro et le temps voulu.
Recommencez ainsi pour chaque position clé, puis sauvegardez le fichier.
Il faut relancer l'application pour qu'elle prenne en compte le nouveau chemin.

- Description rapide des fichiers et des classes

Chaque classe possède son propre fichier, la classe Application est donc définie dans les fichiers « Application.h » et « Application.cpp ».
La classe *Application* est la classe principale.
La classe *MultiCarte* permet de gérer la mosaïque de cartes. Plus précisément c'est elle qui décide des niveaux de détail des données.
La classe *Carte* contient principalement l'algorithme de simplification du relief.
La classe *Chargeur* permet de charger dans un processus différent n'importe quel type de données, ici elle permet de charger les données de relief et les images satellites en arrière plan.
La classe *ChampDeVision* permet l'utilisation de la pyramide de vue.
La classe *Camera* gère la caméra dans l'environnement virtuel.
La classe *Lua* constitue l'interface de communication entre le logiciel et les scripts écrits en Lua.

- Intégrer le moteur dans une autre application

L'intégration du moteur d'affiche est simple. Il suffit de créer une instance de la classe *Application* et d'appeler les fonctions publiques de cette classe au moment voulu.
Le constructeur de cette classe nécessite un pointeur vers un ***CDXUTDialog***, ce qui permet d'afficher des éléments d'interface.
On demande le chargement des données avec la fonction Charger(…) avec comme paramètre un pointeur vers un ***IDirect3DDevice9.***
Le déchargement des données s'effectue avec la fonction Decharger()
La fonction de pré-calcul de l'affichage se fait grâce à la fonction PreCalculer(…) appelée avec en paramètre le temps écoulé depuis la dernier appel à cette fonction.
L'affichage se fait avec la fonction Afficher(…., …..) avec comme premier paramètre un pointeur de ***IDirect3DDevice9*** et comme deuxième paramètre le temps écoulé depuis la dernière image.
Pour finir les divers événements sont gérés par les fonctions :

```
GestionClavier(UINT iLettre, bool bPresse, bool bPresseAlt);
GestionSouris(bool bGauche, bool bDroit, bool bMilieu, int iMoletteDelta, int xPos, int yPos);
GestionInterface(CDXUTControl* pControl, UINT iEvenement, int iControle);
```

- Changer les modes de déplacement

La gestion des modes de déplacement se trouve dans la méthode *FrameMove* de la classe *Camera*.