Rapport de deuxième soutenance

Jérémy ANSELME, Vincent Mirzaian
Rémi Waser, Rémi Weng

12 décembre 2011

Table des matières

1 Introduction **2**

2 Rappels **3**
2.1 L'interface graphique 3
2.2 Pré-Traitement de l'image 4
2.2.1 Filtre moyen (rapide) 4
2.2.2 Filtre moyen 5
2.2.3 Filtre Gaussien 6
2.2.4 Filtre de Sobel 6
2.2.5 Algorithme de Canny 6
2.2.6 Gestion des couleurs et des altitudes 8
2.3 Moteur 3D 9
2.4 Objectifs pour la version finale 11

3 Automap 2.0 **12**
3.1 L'interface graphique 12
3.2 Élimination du bruit 13
3.2.1 Reduction partielle du bruit 13
3.2.2 Miniaturisation de l'image 15
3.2.3 Association des couleurs les plus présentes 15
3.2.4 Intérêt de cet algorithme 16
3.3 Filtre Gaussien variable 16
3.4 Choisir sa hauteur 17
3.5 Quad-Tree 19
3.6 Moteur 3D 21
3.6.1 Optimisations 21
3.6.2 Précalcul des ombres 23
3.6.3 Texture de proximité 23
3.6.4 Skybox et plan d'eau animé 25
3.6.5 Anaglyphes 3D 26
3.7 Site Internet 27

4 Conclusion **28**

Chapitre 1

Introduction

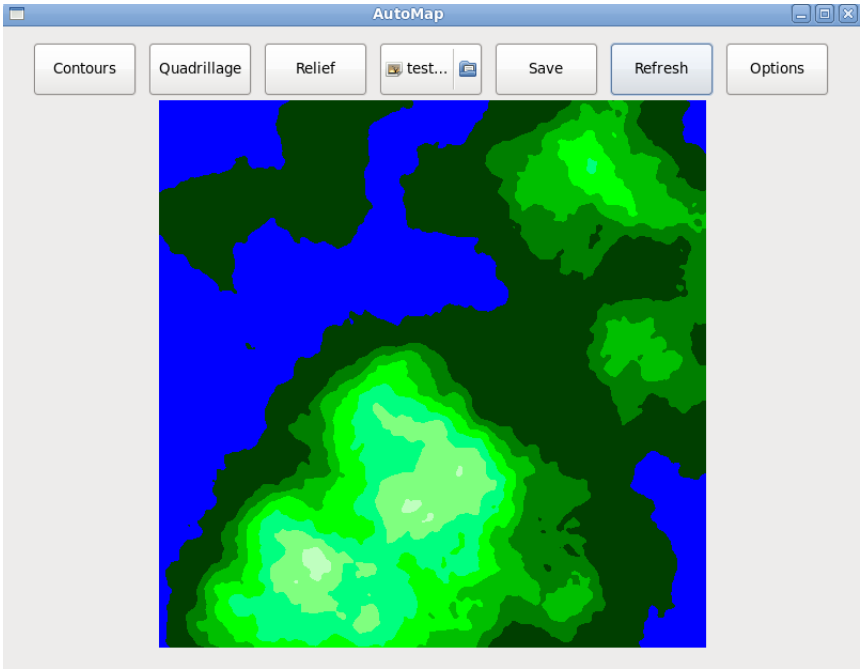
Nous voici enfin sur notre dernière ligne droite dans la réalisation de notre projet de cartographie. Depuis la première soutenance de grandes avancées ont été établies. La team UMAD est donc très heureuse de vous les présenter tout le long de ce rapport. Nous commencerons donc par quelques rappels de ce qui avait été réalisé pour la première soutenance, pour ensuite enchaîner sur les réalisations établies pour cette soutenance. Ce rapport vous détaillera par exemple l'évolution de l'interface graphique, celle des filtres 2D et évidemment celle du moteur 3D qui lui a subit une refonte totale.

Chapitre 2

Rappels

2.1 L'interface graphique

Lors de la première soutenance nous avions une interface très simple que nous avons voulu garder aussi simple pour cette soutenance. Notre interface se composait uniquement d'une fenêtre avec plusieurs boutons dont chacun avaient une fonction bien définie. Par exemple un bouton pour ouvrir une image, un pour quadriller l'image, un pour dessiner les contours, etc ... La bibliothèque utilisée pour l'élaboration de cette interface fut LablGTK que nous utilisons toujours pour cette soutenance.

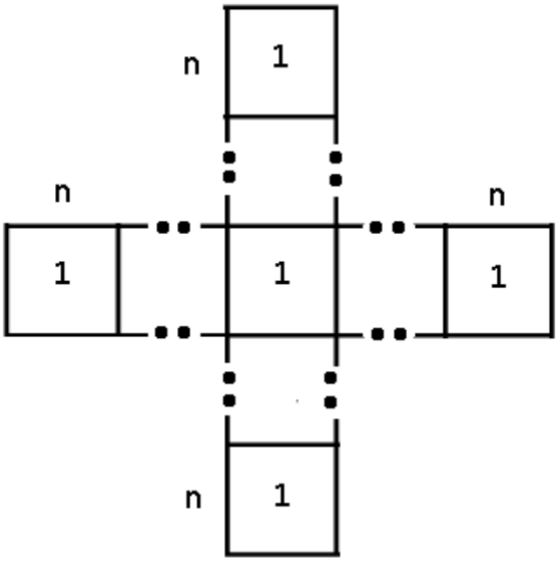


2.2 Pré-Traitement de l'image

Pour la première soutenance nous avions la capacité d'appliquer plusieurs traitements à l'image définie par l'utilisateur.

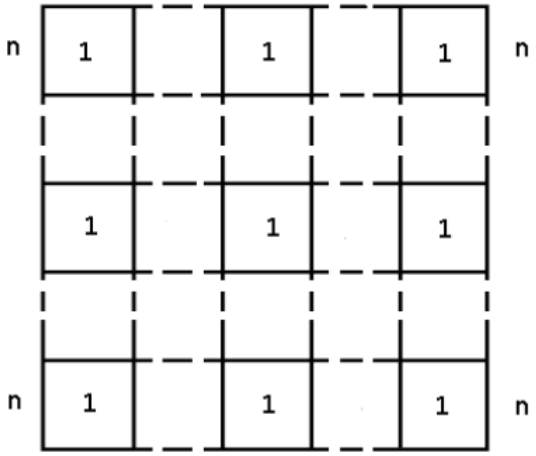
2.2.1 Filtre moyen (rapide)

Pour chaque pixel de l'image, nous faisons la moyenne de la composante rouge de chaque pixel situé à n pixels (pourra être choisi par l'utilisateur) verticalement ou horizontalement (mais pas les deux en même temps) du pixel courant. Puis, nous faisons la même chose pour les composantes vertes et bleues.



2.2.2 Filtre moyen

Nous avions mis en place un autre filtre moyen basé sur le même principe que le filtre précédent mais en faisant la moyenne des pixels situés à n pixels autour du pixel courant (verticalement, horizontalement ou les deux en même temps).



2.2.3 Filtre Gaussien

Le filtre Gaussien utilise le même principe que le filtre moyen mais de façon pondérée. Plus un pixel est proche du pixel courant plus son poids est important alors qu'un pixel éloigné a un poids beaucoup plus faible. Cela explique pourquoi l'utilisateur ne pouvait pas choisir le pas car il fallait regénérer la matrice (dit kernel) d'application.

$\frac{1}{159}$

2	4	5	4	2
4	9	12	9	4
5	12	15	12	5
4	9	12	9	4
2	4	5	4	2

2.2.4 Filtre de Sobel

Le filtre de Sobel permet de calculer approximativement le gradient de l'intensité de chaque pixel. Ceci permet d'indiquer la direction de la variation du clair au sombre (le gradient pointe dans la direction du changement d'intensité). Le gradient est nul lorsqu'on se trouve dans une zone d'intensité constante.

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A \text{ et } G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

Légende : *A représente l'image source (tableau de pixel)*

G_x, le gradient horizontal

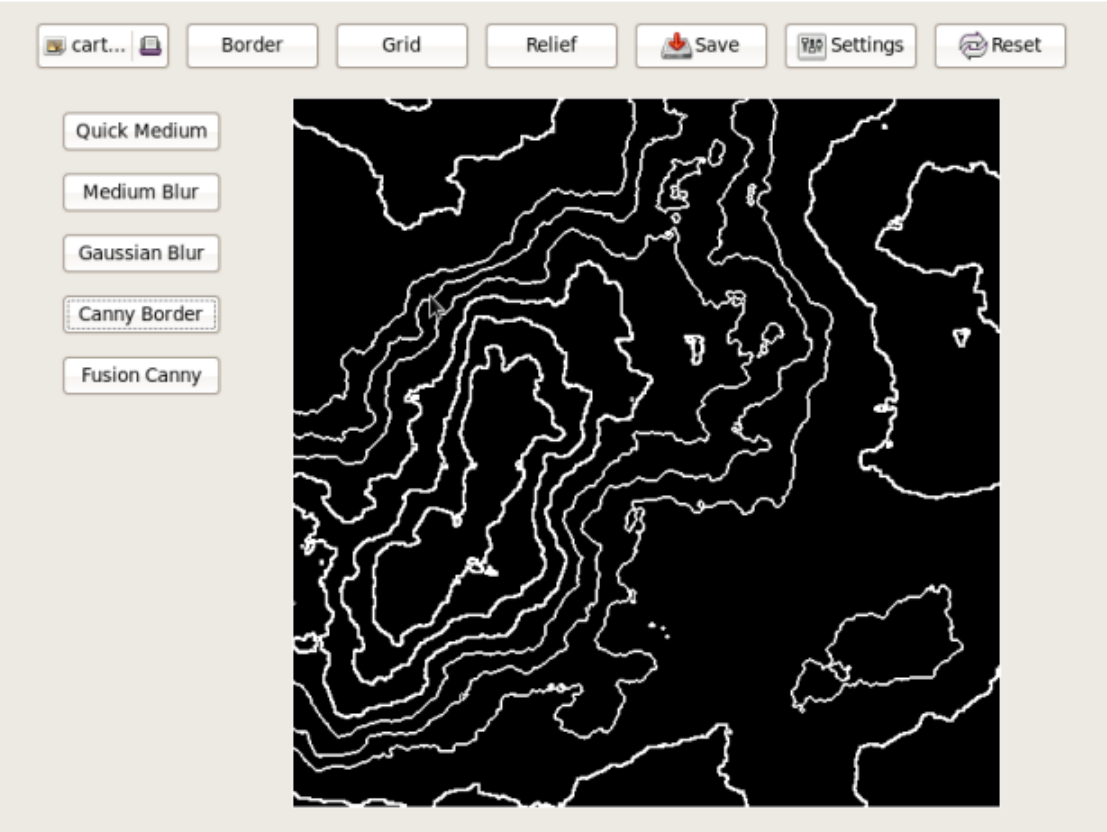
G_y, le gradient vertical

G, la norme du gradient

2.2.5 Algorithme de Canny

Le filtre de Canny se déroule en plusieurs étapes. La première est la réduction du bruit en utilisant le filtre Gaussien (mais rien n'empêche d'utiliser

les filtres moyens, à déterminer en fonction de l'image source et des contours qu'on veut obtenir). La deuxième est le filtre de Sobel qui va calculer le gradient de chaque pixel. Puis, la dernière étape est le seuillage des contours (plus précisément un seuillage par hystérésis) : si l'intensité du gradient est inférieur au seuil.



2.2.6 Gestion des couleurs et des altitudes

Notre objectif final est de reproduire en 3D la carte topologique fournie au programme. Il était donc évident qu'il nous fallait pouvoir définir une altitude pour chaque couleur présente sur la carte. Nous avons d'abord commencé par mettre toutes les couleurs présentes dans une liste. Par la suite, nous avons codé une fonction qui permettait de prédéfinir, dans une liste, la hauteur de chaque couleur.

Pour la première soutenance, nous avions décidé que la première couleur rencontrée sur la carte aura une hauteur nulle, la deuxième une hauteur de 10, la troisième une hauteur de 20 et ainsi de suite. Ensuite il suffisait, de dire au générateur du fichier Wavefront (OBJ), nécessaire à l'ébauche du moteur 3D, de prendre ces altitudes pour la composante Y.

2.3 Moteur 3D

Le moteur 3D est la 2eme plus grande partie de AutoMap : elle se doit d'être intuitive et performante. Par défaut, nos racks sont équipés de la bibliothèque lablGL (un binding OpenGL v1 pour OCaml). C'est donc ce que nous avons utilisé pour la première soutenance. Cependant, nous avons changé pour glMLite pour la deuxième soutenance car il bénéficie d'une version plus récente d'OpenGL (v2) où il est possible d'optimiser considérablement le moteur 3D.

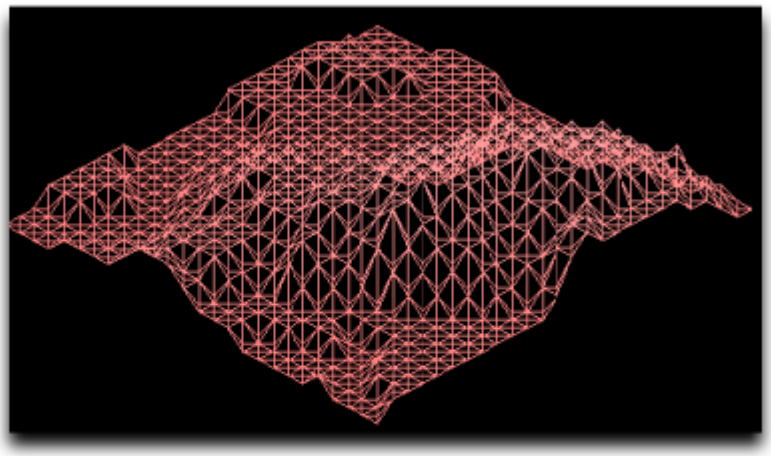
AutoMap affiche dans une fenêtre SDL séparée la visualisation 3D de la carte générée. Sa mise en place a été très rapide, il aura fallu plus de temps pour optimiser certaines méthodes barbares et utiliser des procédures plus ouvertes et donc moins restrictives à notre projet (AutoMap pouvait donc servir à visualiser d'autres modèles que les cartes générée).

En effet, la carte générée par AutoMap est en réalité un fichier Wavefront (OBJ) qui contient une liste indexée de vertex et une liste de faces utilisant cette indexation pour éviter les répétitions de vertex (un vertex peut être attaché à plusieurs faces).

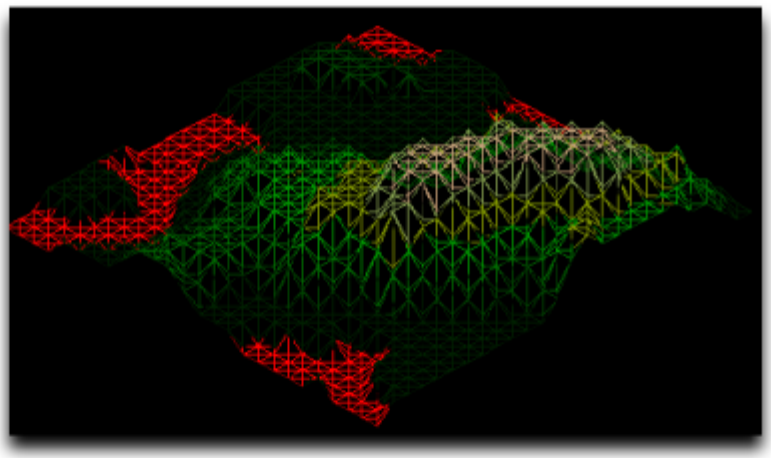
Le moteur 3D lisait donc une première fois le fichier à la recherche des faces (lignes commençant par "f"). Il construisait une liste dynamique de ces dernières tout en essayant de trouver l'index maximal car il déterminait la longueur du tableau à taille fixe et donc à l'indexation rapide de vertex. Une fois la première passe terminée, on fermait le fichier et on créait un nouveau tableau (module Array en OCaml) de la taille précédemment déterminée. On rouvrait le fichier pour capturer tous les vertex en les ajoutant à leur place dans le tableau. Cette méthode de double passe évitait donc tout problème de génération, de fichier altéré, ou encore un ordre différent (définition des faces avant les vertex), tout en garantissant un accès rapide aux vertex lors de l'affichage.

Une fois que la capture des données avait été effectuée, on rentrait dans la boucle principale du programme où la souris et le clavier était utilisés pour manipuler l'espace 3D.

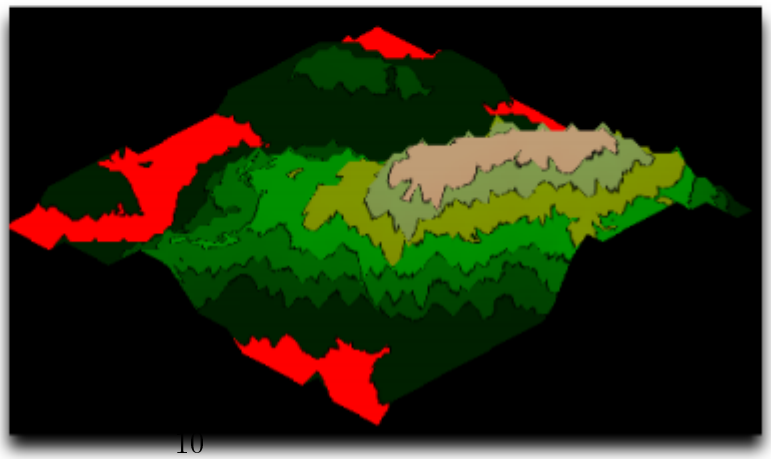
- fil de fer :



- fil de fer texturé (par l'image de l'utilisateur) :



- solide texturé :



2.4 Objectifs pour la version finale

Nous étions en avance sur certains points, tel que la 3D, mais certains aspects du projet nécessitaient d'être beaucoup plus approfondis. En effet, notre interface graphique restait encore assez basique, il fallait donc proposer à l'utilisateur beaucoup plus d'options. Il pourrait alors choisir lui même l'altitude correspondante à chaque couleur de la carte, choisir entre plusieurs modes de triangulation pour la phase d'échantillonnage. Le moteur 3D devait également subir une révision complète pour permettre à l'utilisateur de se déplacer autour du modèle 3D grâce à une caméra à la première personne.

Chapitre 3

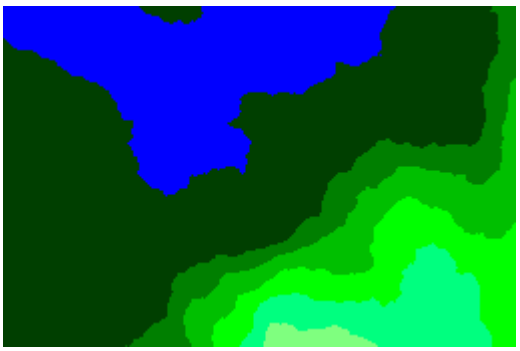
Automap 2.0

3.1 L'interface graphique

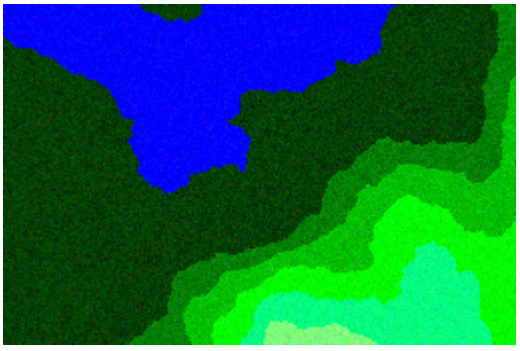
Pour la version finale du logiciel, nous avons partiellement revu l'organisation de l'interface graphique. Comme nous l'avons mentionné dans la première partie de ce rapport, nous avons volontairement souhaité que l'interface reste assez "basique" et simple d'utilisation pour l'utilisateur. Nous avons abandonné l'organisation de l'interface par boutons pour nous diriger vers une barre d'outils où toutes les actions possibles pour l'utilisateur seraient rassemblées. Pour chacun des boutons associes au traitement de l'image, une nouvelle fenêtre s'ouvre pour proposer à l'utilisateur de régler certains paramètres pour ces traitements. Des systèmes de slider ont été ainsi implémentés de manière à ce que le réglage se fasse de la manière la plus simple possible. Lors de la mise en route du moteur 3D, une nouvelle fenêtre s'ouvre avec le modèle 3D de la carte topologique.

3.2 Élimination du bruit

Pour l'échantillonnage de la carte, nous avons voulu dès la première soutenance réaliser un algorithme qui permettrait de réduire voire supprimer l'éventuel bruit que peut contenir une image. Pour rappel, le bruit est une partie (ou la totalité) d'une image dans laquelle se situe des pixels "isolés" et dont la couleur ne correspond pas du tout a la zone dans laquelle il est. Voici un exemple d'image sans bruit :



Et voici la meme mais avec du bruit :



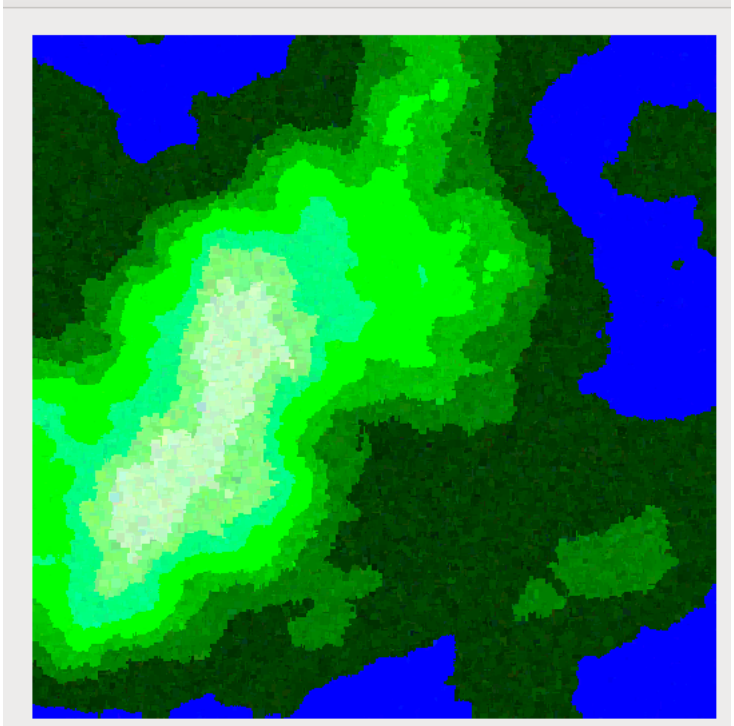
Par manque de temps, nous n'avons pas pu réaliser cet algorithme pour la première soutenance. Voici donc le fonctionnement de celui que vous avons fait pour cette dernière soutenance :

3.2.1 Reduction partielle du bruit

Le bruit étant des pixels "isolés" dans des zones ou ils ne devraient pas être, à l'œil nu il est facile de distinguer la couleur "principale" d'une zone

(par le fait que les pixels soient isolés). Cela est possible car la zone contient plus de pixel de cette couleur que de pixel de bruit.

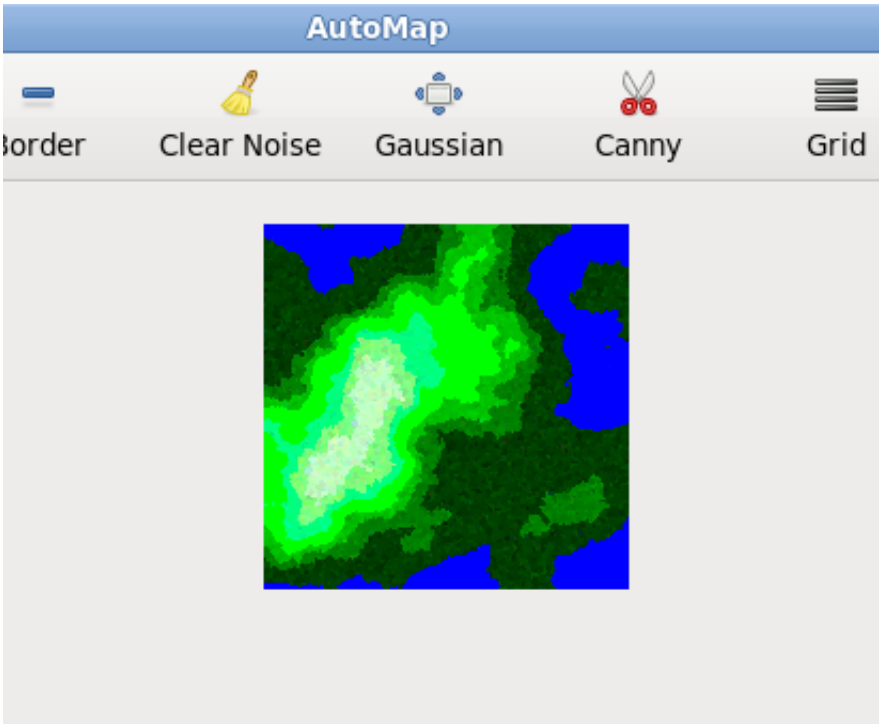
Nous avons, dans un premier temps, réalisé une fonction qui permet pour chaque pixel d'une image de distinguer, parmi tous les pixels autour dans un rayon défini par l'utilisateur (le "range"), la couleur la plus présente. Ainsi, le pixel que l'on parcourt prend cette couleur. On réitère cette opération pour tous les autres pixels de l'image.



Cette première fonction permet de créer un double de l'image source avec beaucoup moins de bruit, mais toujours quelques imperfections comme on peut le voir.

3.2.2 Miniaturisation de l'image

Pour pallier à celles-ci, la deuxième partie de l'algorithme consiste à créer une troisième image qui contiendra uniquement les pixels dont la couleur est la plus présente pour les pixels de l'image. C'est-a-dire que l'on enregistre dans une image les couleurs les plus présentes trouvées à la première étape. Cela permet de faire un système de "zone" et ainsi regrouper petit a petit toutes les couleurs les plus présentes dans l'image. Par conséquent cette troisième image générée est beaucoup plus petite que la première.



Il a fallut trouver un moyen de régénérer l'image taille réelle en utilisant cette troisième image miniature.

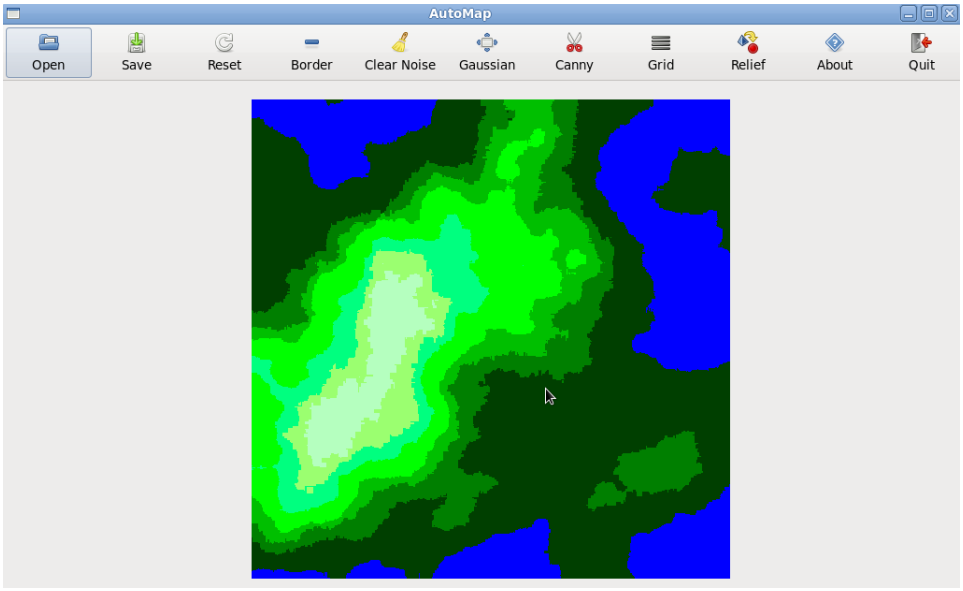
3.2.3 Association des couleurs les plus présentes

La dernière étape consiste à lier les deux étapes précédentes dans une même fonction. En effet, cette troisième fonction va permettre de régénérer l'image taille réelle, mais aussi de détecter intelligemment le moins de couleurs possible que compose l'image d'origine, sans bruit. Pour se faire, on parcourt essentiellement l'image obtenue à la première étape après avoir préalablement généré le tableau contenant les couleurs les plus présentes parmi celles de l'image miniaturisée de la deuxième étape (une double réduction du bruit pour supprimer totalement le bruit, mais avec perte d'informations).

On va comparer un à un chaque pixel de l'image créée par la première étape et les comparer avec les couleurs retrouvées le plus de fois dans l'image

miniaturisée. On prend ainsi la couleur la plus proche parmi les couleurs les plus présentes, et on la remplace dans l'image.

Voici le résultat final :



L'image finale ne contient aucun bruit car nous avons précisé à l'algorithme que le nombre de couleurs maximales de l'image est de 8. Ainsi, il recherche intelligemment les 8 couleurs les présentes sans prendre plusieurs couleurs qui se ressemblent. Les pixels du bruit étant en minorités (car l'image que l'on perçoit avec le bruit est "déchiffrable", on peut voir les contours à l'œil nu), les couleurs de ces pixels sont perdues et ne sont plus reprises dans l'image finale.

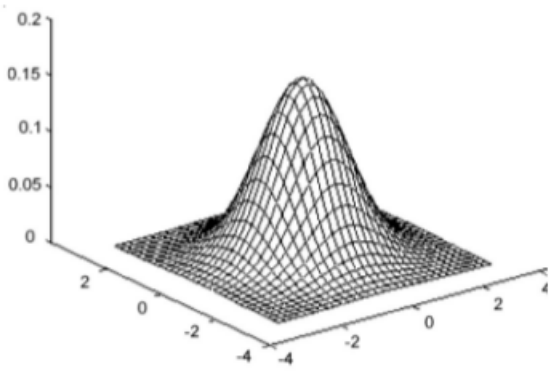
3.2.4 Intérêt de cet algorithme

Cet algorithme permet, à l'aide de paramètres réglables, de réduire voire supprimer le bruit contenu dans une image. Si le bruit est faible, le choix est donné a l'utilisateur de faire une gestion superficielle du bruit. D'un autre côté, si le bruit est en grande quantité, il peut choisir de faire une gestion plus poussée et ainsi obtenir une image plus facile a échantillonner par la suite.

3.3 Filtre Gaussien variable

Afin que l'utilisateur puisse choisir le pas et l'intensité du filtre gaussien, nous avons intégré directement la fonction gaussienne au lieu d'une matrice définie a la main. La fonction est la suivante :

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{(x^2+y^2)}{2\sigma^2}}$$



Si par exemple $\sigma = 0.8$ on a le filtre 3×3 suivant

$$\begin{array}{|c|c|c|} \hline G(-1,-1) & G(0,-1) & G(1,-1) \\ \hline G(-1,0) & G(0,0) & G(1,0) \\ \hline G(-1,1) & G(0,1) & G(1,1) \\ \hline \end{array} \simeq \frac{1}{16} \cdot \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$$

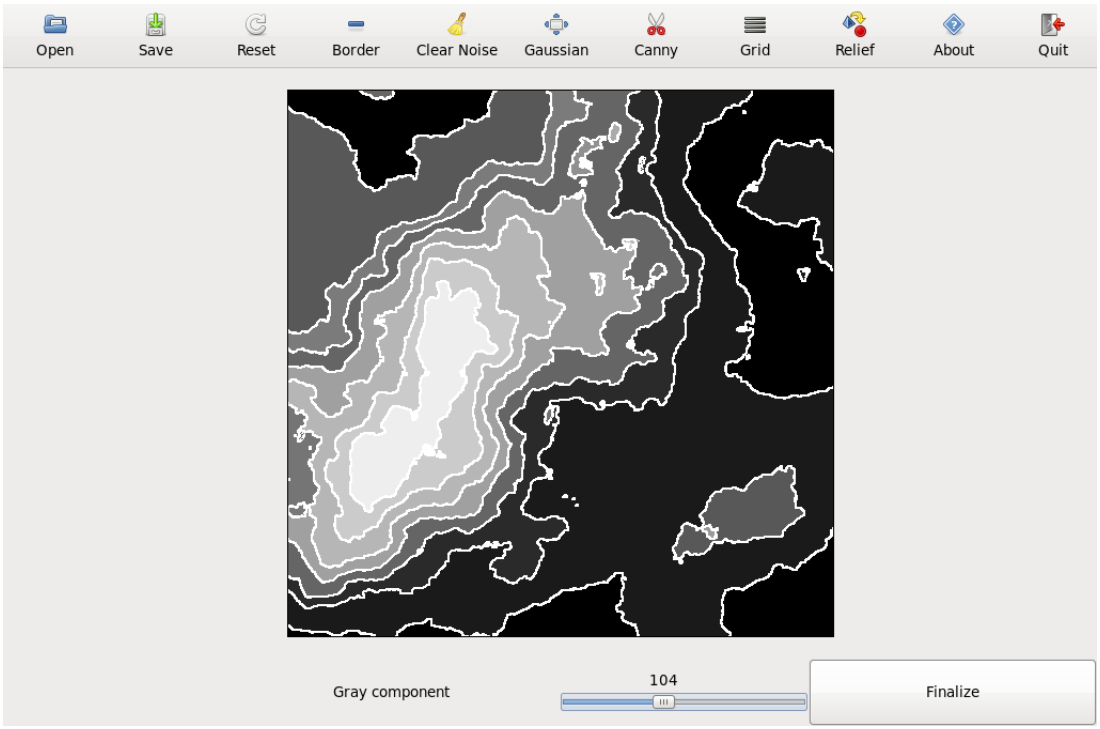
et $\sigma = 1$ pour un filtre 5×5 donne environ

$$\frac{1}{300} \cdot \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 6 & 4 & 1 \\ \hline 4 & 18 & 30 & 18 & 4 \\ \hline 6 & 30 & 48 & 30 & 6 \\ \hline 4 & 18 & 30 & 18 & 4 \\ \hline 1 & 4 & 6 & 4 & 1 \\ \hline \end{array}.$$

3.4 Choisir sa hauteur

Nous avons donné à l'utilisateur la possibilité de choisir la hauteur des zones de la carte. Pour mettre en place cette fonctionnalité, nous avons profité du fait que le Quad Tree utilise le niveau de gris. Donc, la solution que nous avons trouvé est de permettre à l'utilisateur de choisir la hauteur en niveau de gris directement (plus la couleur est foncée/noire, plus la hauteur de la zone est basse et à l'inverse, plus la couleur est claire/blanche, plus la zone se situe à une hauteur élevée).

Pour que l'utilisateur puisse définir lui-même la hauteur des zones et que cela reste intuitif, nous avons intégré dans l'interface graphique la possibilité de choisir le niveau de gris (donc la hauteur de la zone). L'utilisateur doit simplement cliquer sur la carte la zone avec le niveau de gris qu'il désire.



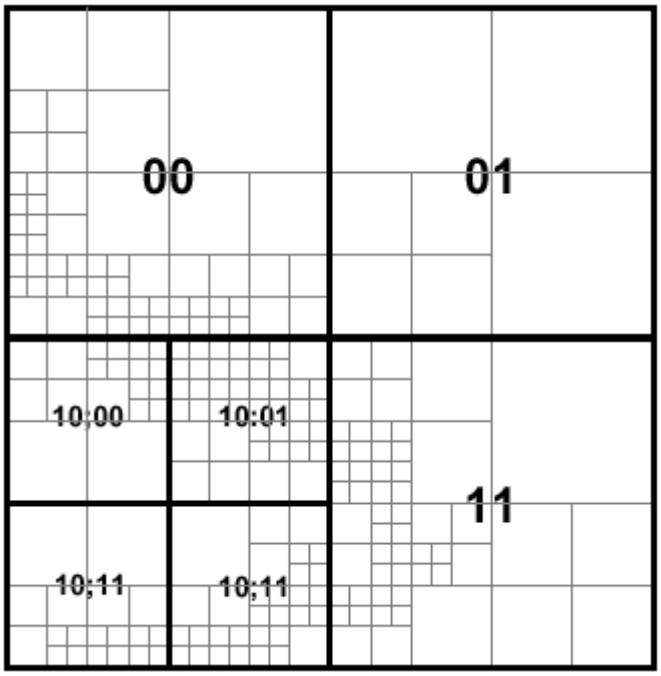
Cette fonctionnalité utilise l'image générée par l'algorithme de Canny puisqu'elle ne contient que des contours. Il nous a simplement suffi de coder une fonction de remplissage de couleur dans une zone en fonction des contours mis en évidence par l'algorithme de Canny.

Pour finir, il faudra connaître la position de la souris et de l'image affichée dans l'interface (Pour détecter quand l'utilisateur clique sur l'image et connaître la position de la souris dans l'image afin de remplir la bonne zone en fonction du niveau de gris). Ensuite si la couleur de la zone cliquée existe autre part, ces dernières sont également remplies du même niveau de gris.

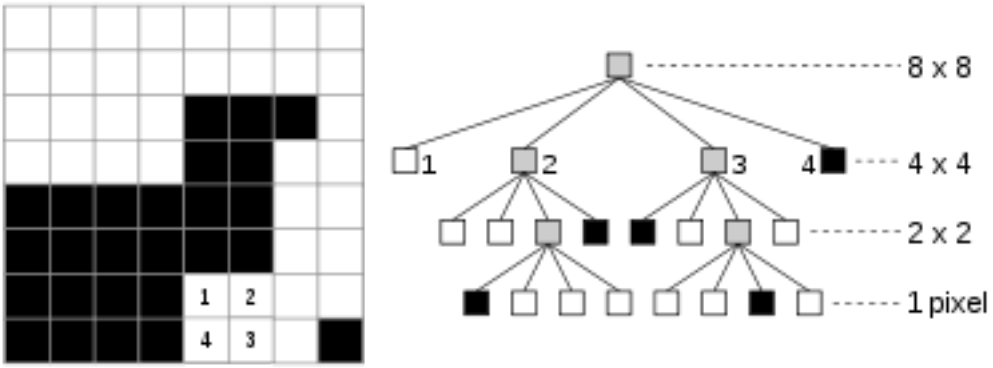
3.5 Quad-Tree

Du fait que la triangulation de la première soutenance ne faisait que découper l'image en zones identiques suivant un pas défini par l'utilisateur, il était indispensable de partir sur une nouvelle technique de triangulation. En effet, certaines zones étaient plates et il n'était pas nécessaire d'avoir une densité de maillage importante dans ces zones. C'est pourquoi nous avons opté pour la triangulation avec la technique du Quad-Tree.

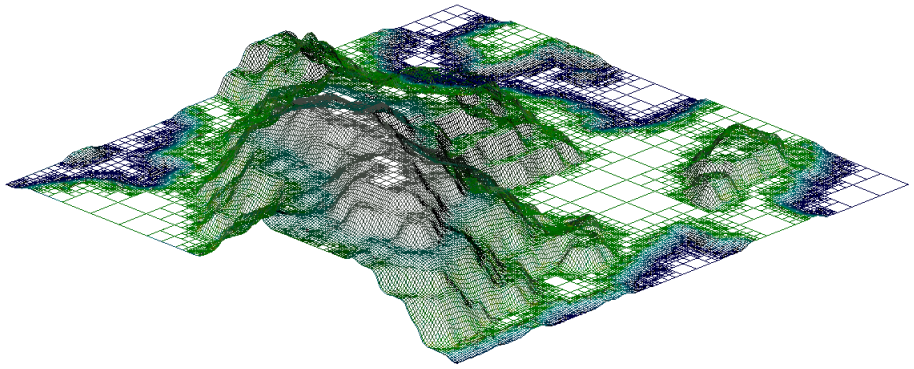
En quoi consiste le Quad-Tree ? Il s'agit d'une technique de triangulation qui peut paraître compliquée mais qui en réalité reste relativement simple à appréhender. Il en existe plusieurs variantes. Nous avons utilisé la plus simple et la plus rapide. Tout d'abord, on regarde tous les pixels de l'image jusqu'à trouver une discontinuité dans les valeurs des pixels. Si on trouve un pixel trop différent du premier pixel de l'image, on divise l'image en 4 zones de même taille. Il suffit alors de recommencer le processus sur ces 4 nouvelles zones jusqu'à atteindre soit le niveau de subdivision maximal, soit le stade du pixel (zone de 1*1 pixels).



Le Quad-tree peut être représenté conceptuellement sous forme d'arbre où chaque nœud est une division, et chaque feuille une face. Étant donné que cette technique doit impérativement être récursive, les appels récursifs sont très nombreux pour des images haute définition avec beaucoup d'irrégularités, nous avons inséré l'écriture des faces en feuille, et l'écriture des points après la génération du maillage. De plus, la création des points est accompagnée d'une table de hachage permettant de traiter dans un temps record les points en double.

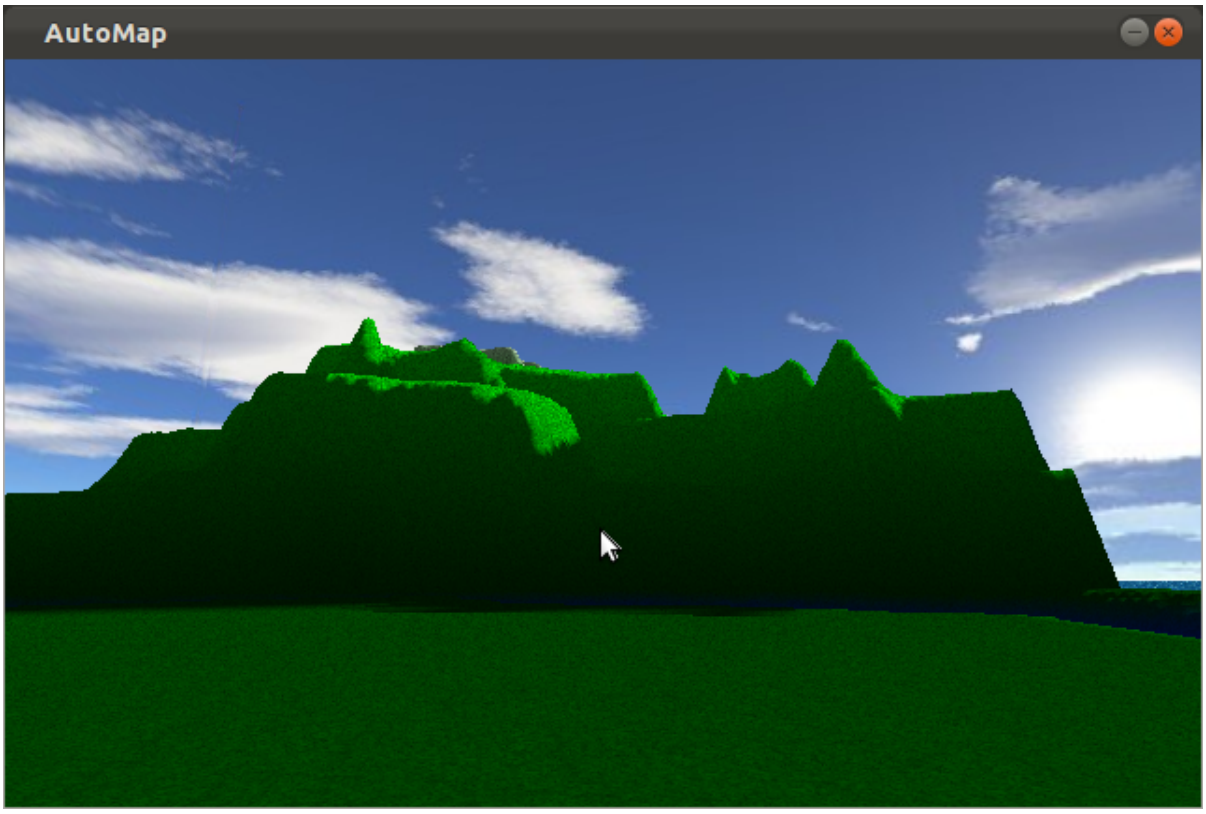


Voici une génération concrète de notre carte :



3.6 Moteur 3D

Nous voici enfin à la partie la plus passionnante de ce rapport et de ce qui constitue le cœur du programme. En effet, le moteur 3D a eu droit à une refonte totale. Nous avons décidé de partir sur de nouvelles bases pour que notre moteur 3D soit encore plus puissant.



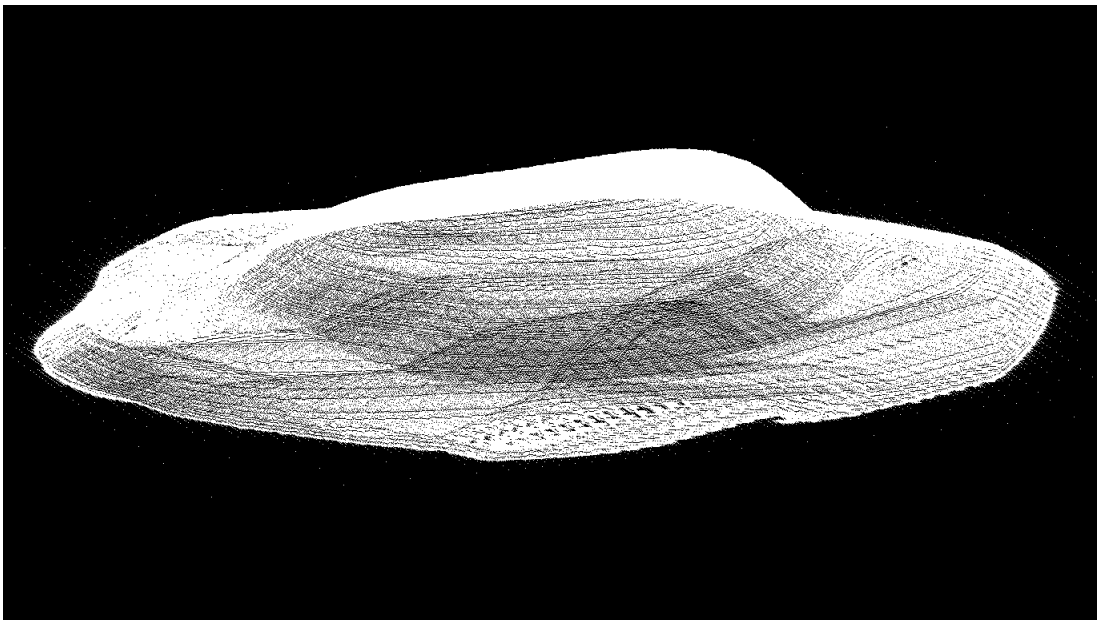
3.6.1 Optimisations

Vertex Buffer Object

LablGL a plein de qualités, celles de OpenGL 1.X. Or cette version d'OpenGL ne supporte pas une optimisation absolument géniale qui s'appelle les Vertex Buffer Object (VBO). Les VBO consistent au fait de n'envoyer qu'une seule fois le tableau de point et de faces (ou autres comme les normales, textures etc ...) à la carte graphique. Ces données sont donc allouées dans un espace mémoire de la carte graphique et cette dernière va pouvoir les récupérer sans laps de temps.

Nous nous sommes penchés sur glMLite car il supportait les VBO et avons converti tout le code vers glMLite afin de pouvoir utiliser les nouveautés de OpenGL 3.X. Suite à cela, nous avons ajouté la gestion des VBO et là ce fut le comble ! Nous avons dorénavant la possibilité de monter jusqu'à 10

millions de points et 8 millions de facettes sans un seul lag. Par la suite nous avons rajouté les textures et la gestion des normales et nous avons vu ces performances baisser quelque peu, seulement, cela reste bien mieux que notre précédente limite de 200 000 points avec LablGL.



Tableaux VS Listes

Pour la première soutenance nous utilisions toujours des listes pour stocker les faces car nous n'avions aucun moyen de connaître à l'avance le nombre de faces avant la lecture du fichier pour créer un tableau à taille fixe. Or l'utilisation des listes en OCaml est limitée à environ 100 000 éléments pour les algorithmes récursifs sinon une erreur "Segmentation Fault (Looping recursion ?)" est très probable.

Nos modèles générés par le nouveau QuadTree étant de plus en plus lourds (mais optimisés) il fallait repousser cette limite. Nous avons donc pensé à une manière de se souvenir du nombre de points et de faces avant de commencer à remplir les tableaux.

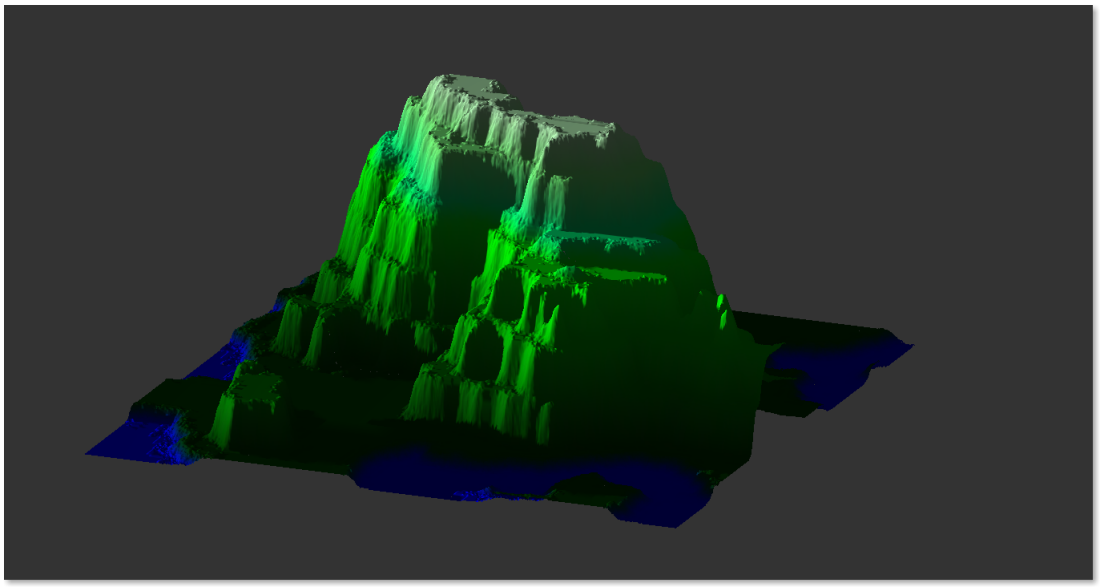
À la fin de la génération du QuadTree nous connaissons le nombre de points et de faces créés et nous pouvons donc l'écrire en commentaire à la fin du fichier (nous n'avons aucun moyen d'écrire au début sans devoir réécrire tout le fichier).

Lors de l'ouverture du fichier par le moteur 3D, nous allons récupérer ces deux nombres directement à la fin du fichier si et seulement s'ils existent. Dans le cas contraire, on revient au début du fichier et on compte chaque face

et chaque point. Cela offre donc un stockage optimisé, rapide et important pour les faces et les points pour des fichiers générés par AutoMap, mais également offre une compatibilité pour d'autres fichiers Wavefront (OBJ).

3.6.2 Précalcul des ombres

OpenGL ne fourni pas la possibilité de calculer automatiquement les normales des faces, ni même la possibilité une fois ces normales pré-calculées de tracer des ombres. C'est compréhensible et il existe pleins de manières de le faire. De notre côté, nous avons profité du calcul des normales des faces pour insérer un algorithme de raytracing :

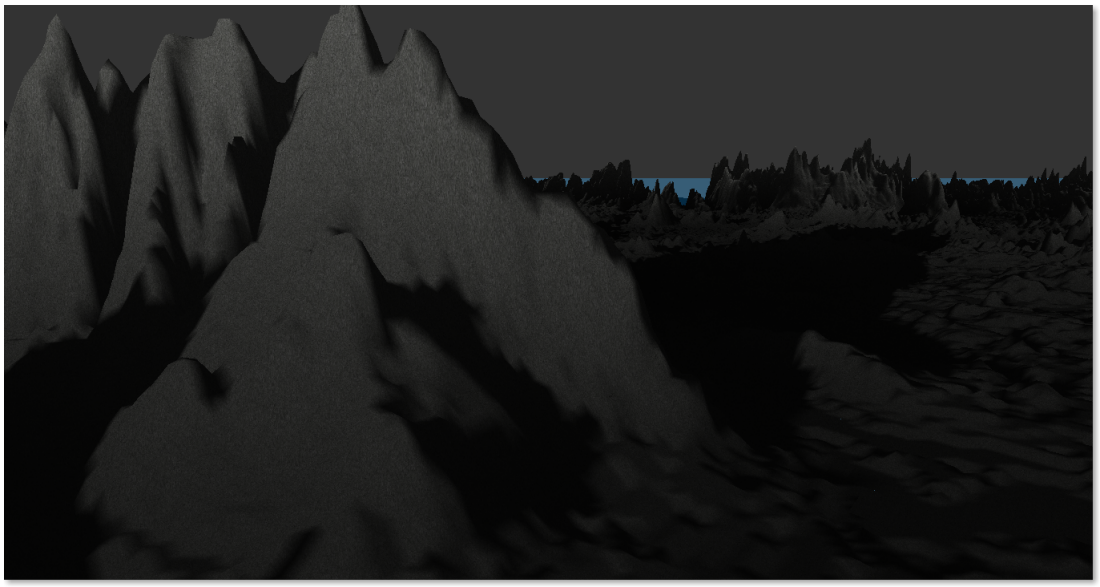


3.6.3 Texture de proximité

Une fois le modèle chargé il est possible de s'en approcher de près, de tellement près qu'on en perd le plaisir car la texture de base est en haute définition pour le modèle entier, mais pas pour l'échelle que l'on utilise pour regarder le sol d'aussi près.

C'est pourquoi nous avons rajouté une texture de proximité mélangée avec la texture du modèle. Seulement, lors de nos recherches sur le sujet nous avons été dans l'impossibilité de trouver la fonction glClientActiveTexture dans le binding glMLite. Nous rappelons que nous avons opté pour ce binding car c'est le seul qui support des VBO pour Ocaml. Or ce dernier ne possède pas cette fonction.

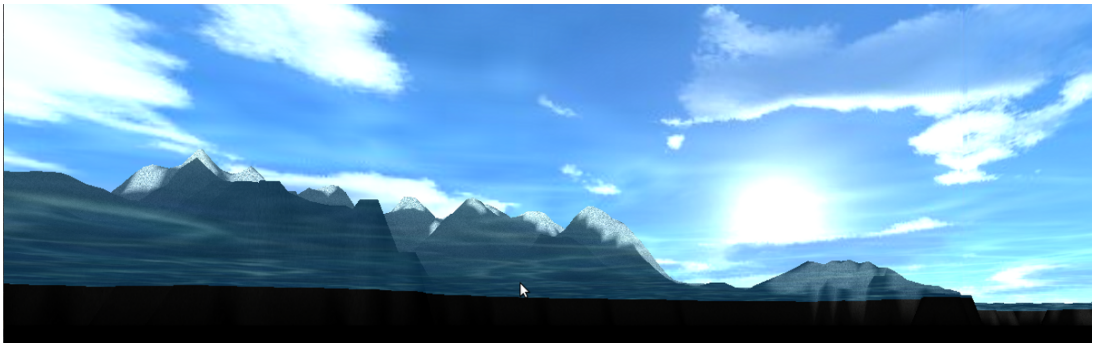
Nous avons donc été obligé de modifier légèrement glMLite pour rajouter cette fameuse fonction, sans ça, le multi-texturing avec les VBO aurait été impossible ! Heureusement que ce genre de binding consiste simplement à lister les équivalents des fonctions entre le C et OCaml

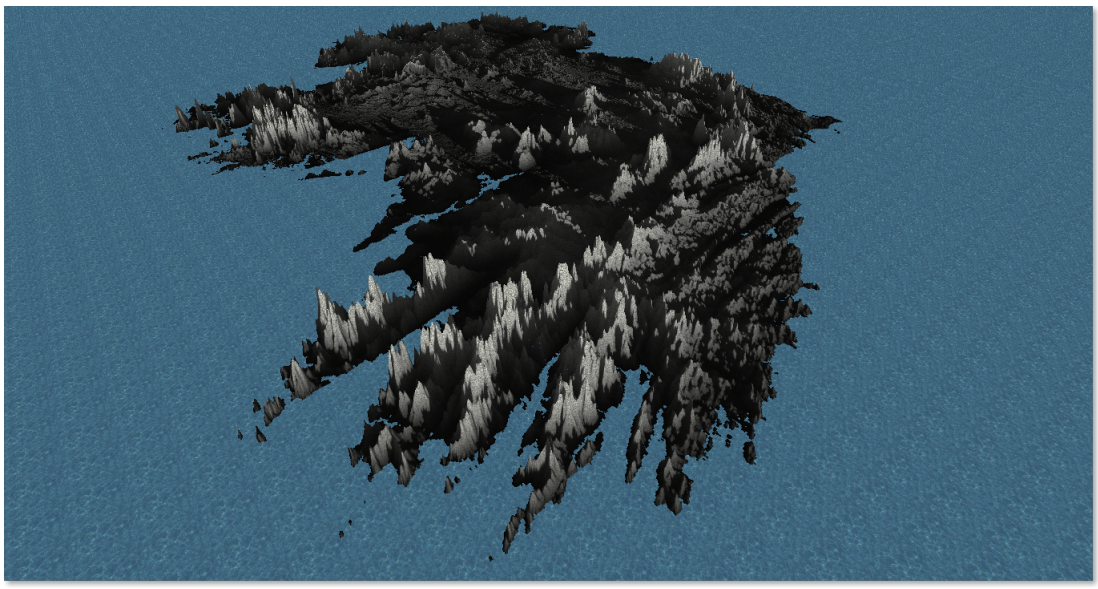


3.6.4 Skybox et plan d'eau animé

Nous avons rajouté une skybox aussi simplement que de rajouter un cube texturé non influencé par la lumière.

Par ailleurs, nous avons rajouté un plan d'eau animé. Cela consiste simplement à avoir un plan texturé non opaque de très grande taille et possédant des coordonnées de texture changeant au cours du temps.

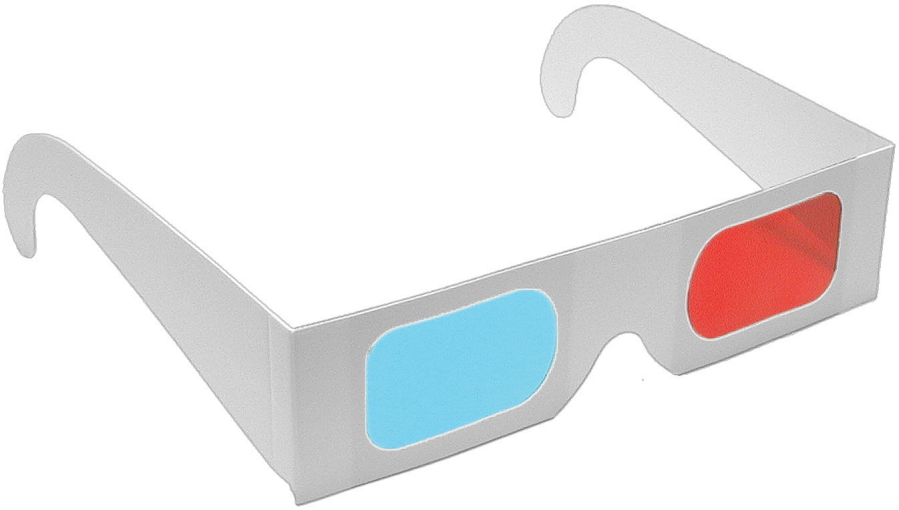


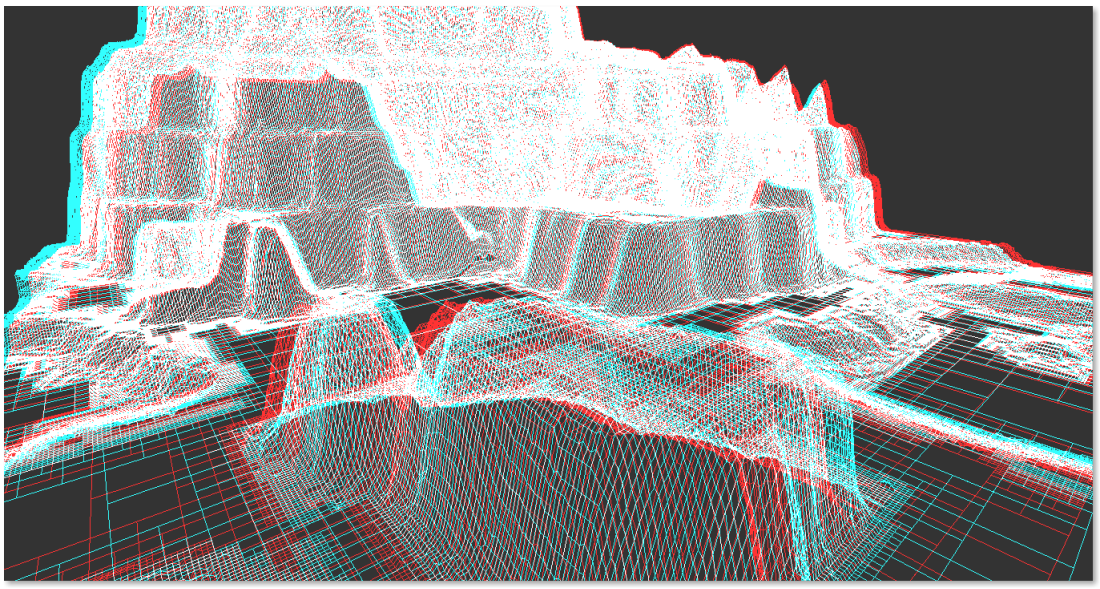


3.6.5 Anaglyphes 3D

Afin de rendre AutoMap attractif nous avons rajouté une fonctionnalité assez intéressante, celle d'afficher la scène 3D de manière à ce qu'elle soit perçue en relief par l'utilisateur portant des lunettes rouges/cyan.

Le principe est simple, nous rendons deux fois la scène depuis deux endroits différents simulant l'écart oculaire entre l'œil droit (cyan) et l'œil gauche (rouge).





3.7 Site Internet

Grâce à l'avance que nous avons pris sur le projet, nous avons pu en profiter pour changer le design du site (disponible à l'adresse suivant : `http://umad.fr.nf`) qui est maintenant beaucoup plus attractif et donne envie d'en savoir plus.

On trouve tous les informations du projet, les membres du groupe, l'actualité, l'avancement du projet etc... On peut maintenant télécharger directement sur le site le projet pour pouvoir tester.

Chapitre 4

Conclusion

Nous achevons aujourd'hui ce projet qui nous aura demandé plus de 2 mois de travail. Durant ces deux mois de travail, chacun des membres du groupe a pu mettre sa pierre à l'édifice et a pu enrichir ses connaissances aussi bien sur le plan technique de la programmation que sur sa culture générale en informatique. Nous sommes donc aujourd'hui heureux d'avoir pu concevoir un tel programme donc les bénéfices futurs ne seront que plus importants.