École polytechnique de l'Université de Nantes

–

Université Libre de Bruxelles

Stage de quatrième année

Maillage en temps réel de nuages de points enregistrés par le Kinect

Étudiant :
Olivier Catry
Polytech' Nantes

Encadrant :
Benoît Penelle
Laboratoire LISA, ULB

29 juillet 2011

Preface

Ce rapport présente le projet que j'ai réalisé au cours de mon stage de quatrième année d'informatique à l'École polytechnique de l'Université de nantes.

Celui-ci s'est déroulé durant deux mois au LISA, à l'Université Libre de Bruxelles, sous la tutelle de Benoît Penelle, que je remercie pour m'avoir proposé ce sujet. Je remercie également Nadine Warzée ainsi que toute l'équipe du laboratoire pour leur accueil chaleureux.

Présentation du laboiratoire
Le laboratoire de l'Image Synthèse et Analyse (LISA) fait partie du service LIST, faculté des Sciences appliquées/école polytechnique - Technologie de l'information. Dans le domaine de l'analyse d'image et de la reconnaissance de formes, le LISA aborde des problèmes 2D et 3D de segmentation (via des méthodes originales de morphomathématiques et d'apprentissage) et de suivi d'objets mobiles dans des séquences d'images. Les algorithmes développés sont adaptés à de nombreuses applications issues de domaines complètement différents, tels que : - biomédical : suivi cellulaire in vitro, caractérisation de lésions cutanées pigmentées, analyse de marquages immunohistochimiques, étude de l'emphysème pulmonaire et du cartilage articulaire, recalage non rigide inter-modalité ; - télé-détection : segmentation automatique d'images, reconnaissance d'objet, aide à la mise à jour de carte ; - applications industrielles : accélération matérielle de la reconnaissance d'empreintes digitales, analyse automatique de trafic routier. Les activités de recherche liées à l'image de synthèse et la réalité virtuelle sont orientées vers des applications médicales et des applications "temps réel" telles que : - médecine : réduction de fracture en réalité virtuelle, chirurgie du foie assistée par ordinateur ; - archéologie : reconstitution virtuelle d'objets sur base de fragments ; - visibilité dans des environnements complexes ; - interface gestuelle pour la navigation dans des environnements virtuels ; - modélisation 3D de divers phénomènes, tels que la croissance d'arbres, les collisions d'objets, le mouvement de cheveux, les déformations de tissus, l'eau, le feu ,... `http://www.ulb.ac.be/rech/inventaire/unites/ULB367.html`

Table des matières

1 Introduction

Le projet de Benoît Penelle concernant le suivi et l'analyse de mouvement est décrit de la façon suivante : "*Le but de ce projet est de développer une interface non-intrusive (sans capteurs) et intuitive pour différents types d'applications. Le système se base sur la capture et l'interprétation de la position et des mouvements de l'utilisateur grâce à l'analyse d'images fournies par une caméra 3D.*"

Seulement, l'affichage des images fournies par la caméra 3D était constitué de nuages de points. Afin d'obtenir l'affichage d'une forme géométrique représentant l'enregistrement de la caméra 3D, un projet de stage ayant cet objectif a été proposé. Ce rapport présente ce projet dit de "Maillage en temps réel de nuages de points enregistrés par le Kinect".

Pour gagner en efficacité, le rapport présente en tout premier lieu, et de façon vulgarisée, les enjeux du projet et directement une explication de ce qui a été produit.
Puis, pour l'intérêt purement informatique du projet, le détail est fait sur la réalisation : Conception, implémentation et tests des performances.
Pour garder une trace des explications sur le code et faciliter son intégration pour de nouvelles versions du projet, une documentation technique a été réalisée.
Enfin, un point est fait sur la façon dont ce stage a été organisé.

2 Enjeux

2.1 Fait : affichage du nuage de point

Le Kinect enregistre une double information. La première information est la même qu'enregistre une caméra classique, c'est-à-dire une image en couleur de son champs visuel. La seconde information est une image des profondeurs, soit la distance à laquelle se situe chaque point enregistré par rapport au point de vue. C'est à partir de ces deux images que le nuage de point est généré. L'une des subtilité du Kinect est de quantifier la distance selon un pas plus grand à mesure que l'on s'éloigne du point de vue. Ainsi, les éléments proches ont une précision sur la distance supérieure aux éléments lointains.

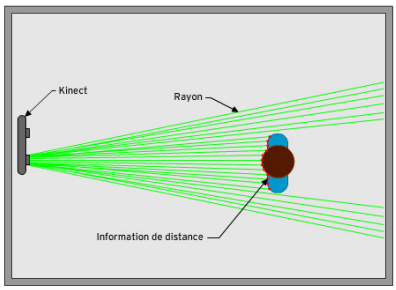


FIGURE 1: Fonctionnement du Kinect [2]

FIGURE 2: exemple d'image de profondeurs et image de couleurs enregistrés par le Kinect

L'affichage du nuage de points est réalisé de la façon suivante : Une classe contient les attributs relatifs à chaque nuage de point enregistré par le kinect. Pour chaque point, elle renseigne ses coordonnées dans l'espace ainsi que les valeurs de ses trois composantes rouge, vert et bleu. Ces attributs sont récupérés au moment de l'affichage et les points sont affichés en fonction de ceux ci.

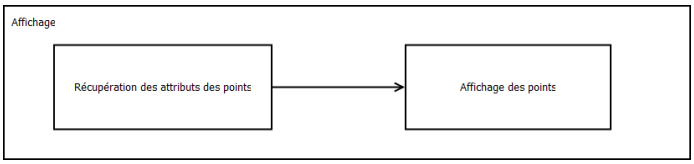


FIGURE 3: Schéma décrivant le fonctionnement de l'affichage du nuage de points

Le rendu est acceptable vu d'une certaine distance. Mais dès que l'on approche la caméra, on peut constater des espaces entre les points. Ceci pose alors un problème dans l'optique d'obtenir un modèle proche de la réalité.

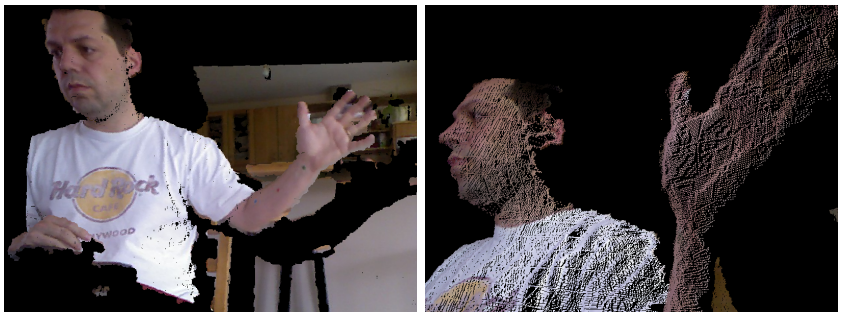


FIGURE 4: Affichage depuis deux points de vues différents du nuage de points

2.2 Premier objectif : affichage du mesh

Un algorithme de maillage a donc été réalisé, il vient se greffer au programme d'origine et remplace l'affichage du nuage de points par l'affichage d'un *mesh*.

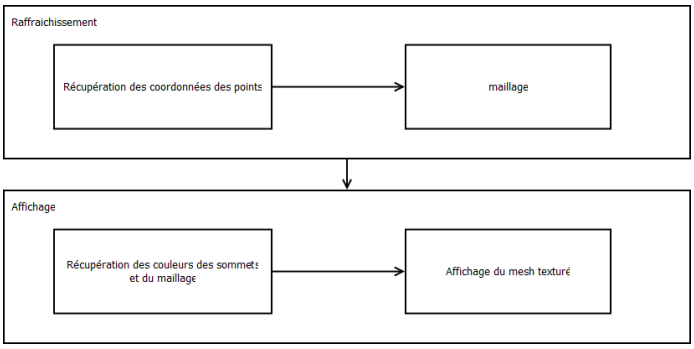


FIGURE 5: Schéma décrivant le fonctionnement de l'affichage du mesh

Le rendu ne présente désormais plus de points mais une surface continue, dans la mesure où des points le permettent. Les points trop éloignés ne sont pas reliés. La méthode pour définir l'éloignement a été fixée par une distance minimale de reliure paramétrable par l'utilisateur. Nous pouvons noter une imprécision dans la formation du mesh, qui est due à un procédé d'optimisation qui est une conséquence du second objectif : le temps réel.

FIGURE 6: Affichage depuis deux points de vues différents du mesh

2.3 Second objectif : temps réel

La solution retenue pour cet objectif puise l'idée dans un procédé bien plus complexe et plus vaste dont l'optimisation réside grossièrement dans le fait d'utiliser moins de polygones. Le découpage est effectué selon un pas défini, puis au niveau des contours, il est de nouveau effectué avec un pas plus petit. Le découpage est recommencé jusqu'au pas défini comme étant le plus petit.

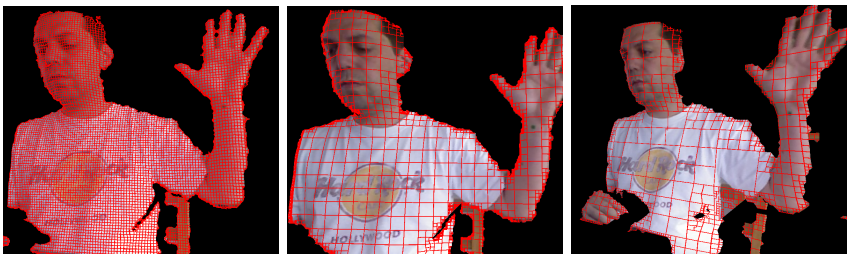


FIGURE 7: découpage du mesh selon trois paramétrages différents, et affichage des zones de découpage en rouge. Le troisième découpage est le plus optimisé des trois.

Frames	Time (ms)	Min	Max	Avg
260	30000	8	9	8.667

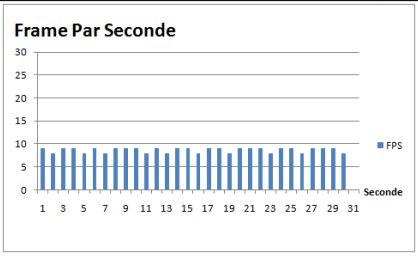


FIGURE 8: Benchmark (FRAPS) sans optimisation

Frames	Time (ms)	Min	Max	Avg
767	30000	24	27	25.567

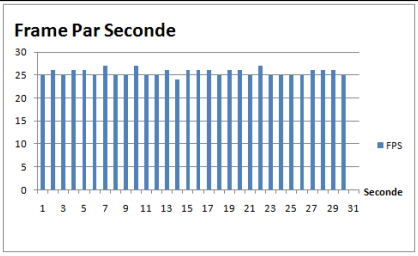


FIGURE 9: Benchmark (FRAPS) avec optimisation : pas maximal de 40 et minimal de 5

3 Réalisation

3.1 Recherche

L'algorithme utilisé pour faire le maillage utilise la structure de double-image. Il se déroule de la façon suivante :

1. Dans l'image des distances, on prend un point P1. On prend son voisin de droite P2. On prend également son voisin du dessous P3. Enfin, on prend son voisin en bas à droite P4. On a alors quatre coordonnées de quatre points qui forment un carré d'étude.
2. Dans ce carré d'étude, on compare les valeurs de distance des quatre coins. La valeur absolue de la différence donne la distance relative de chaque coin deux-à-deux. Il existe alors de nombreuses configurations possibles. Selon les configurations, le découpage est fait de façon à former soit un triangle, soit deux triangles (cas où il y a distance proche entre tous les coins), soit aucun découpage résultant sur la formation d'aucun polygone, dans le cas où les distances n'offrent pas la possibilité d'en former.
3. On recommence de la même manière en couvrant tous les carrés d'étude que peut contenir l'image.

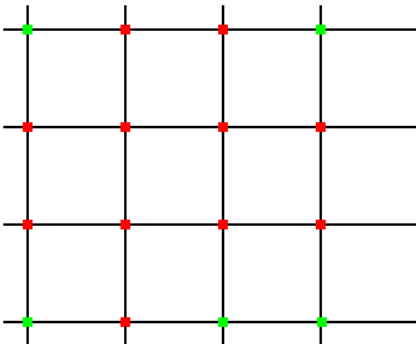


FIGURE 10: Sur cet exemple, plusieurs points sont présents sur l'image des distance. Les points colorisés en rouge sont d'une distance égale du point de vue, de même que pour les points colorisés en vert. La distance relative entre le rouge et le vert est trop grande pour accepter la reliure.

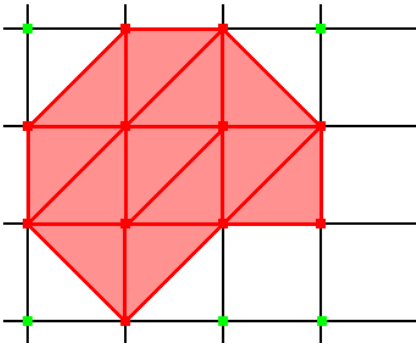


FIGURE 11: Chaque carré est alors traité un-par-un et découpé en triangles en suivant la disposition des reliabilités entre les coins.

Il existe d'autres moyens de réaliser un tel découpage. Par exemple, une autre idée a été d'utiliser le GLSL (Graphic Library Shader Langage) basé sur un découpage depuis un *mesh template*. L'intérêt est d'utiliser une grille déjà divisée en triangle, et de relier deux à deux les sommets en suivant leur adjacence sur la grille. On travaille alors sur une 6-séparation contrairement à la 8-séparation précédente qui permettait donc d'épouser mieux la forme.

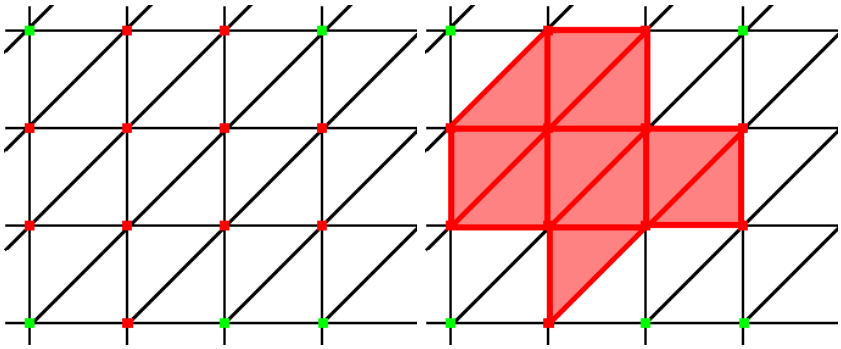


FIGURE 12: Comme sur l'exemple précédent, le découpage est fait selon une grille précalculée. Le mesh épouse moins bien la forme

Cette méthode présente toute de même un avantage puisqu'elle utilise les techniques de *Vertex Shader* et de *Geometric Shader* qui permettent de programmer directement sur le *pipeline* de la carte graphique, et donc de gagner en performance. Ceci est détaillé dans la partie traitant des solutions alternatives.

Après une première implémentation, de nouveaux problèmes se sont posés, dont un nécessitant une nouvelle recherche. Comme en témoigne le benchmark montré dans la partie Enjeux, le *framerate* est très bas. Neuf image par seconde n'est pas suffisant pour être du temps réel. Le programme a été limité à 30 FPS (Images ou Frames Par Seconde) et il faut que l'affichage du mesh puisse tourner au moins à 24 FPS.
Une idée est venue du projet multimédia de 4eme année dont une partie portait sur le *normal mapping*. Une technique abondamment utilisée aujourd'hui dans le domaine du jeu-vidéo qui consiste à réduire le nombre de polygones d'un mesh tout en lui conservant les propriétés de *texturing* et de *bump mapping* telles qu'elles seraient appliquées au mesh polygoné comme à l'origine.

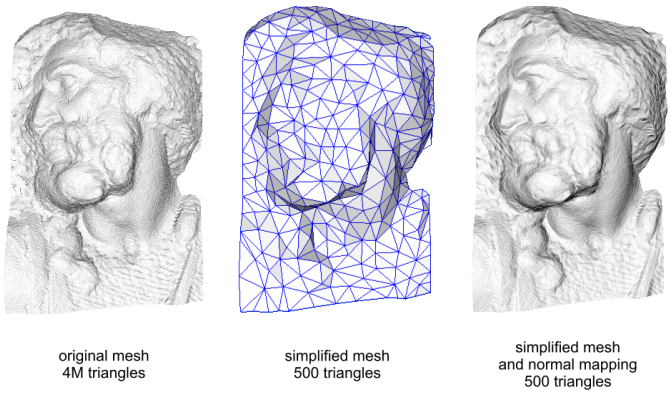


FIGURE 13: Normal mapping appliqué à un mesh permettant de passer de 4 millions à 500 triangles [3]

En reprenant le principe de diminution du nombre de polygones tout en conservant simplement la texture d'origine, il est possible d'optimiser le rendu du mesh. Deux façons de faire ont été trouvées, et l'une d'entre elles a été retenue :

1. La première idée était une étude fréquentielle de l'image des profondeurs. Sur les basses fréquences, on forme des carrés en utilisant moins de points, c'est-à-dire avec un pas plus grand. Sur les hautes fréquences, le pas diminue. Le découpage est alors plus fin lorsque la variation des distances est importante.
2. La seconde idée rejoint la première dans son principe mais pas dans sa réalisation. Le découpage est directement fait avec un pas de grande taille, ce qui génère un premier mesh découpé de façon grotesque. Un second découpage est effectué avec un pas moindre et concerne uniquement les zones qui n'ont pas encore été polygonées. L'opération se répète jusqu'à atteindre un pas minimal choisi. C'est cette idée qui a été retenue car c'est celle qui était la plus aboutie.

(a) première itération (b) dernière iteration moins la première (c) dernière itération

FIGURE 14: Etat visuel du mesh au cours de l'algorithme

L'idée pouvait fonctionner selon les attentes, mais contrairement au résultat attendu montré ci-dessus, il réside un effet secondaire à une telle méthode : les coutures entre polygones. Le découpage en carrés grossiers réalise une interpolation linéaire dans le maillage, qui ne passe alors pas par les points qui ne sont pas pris en compte à cause du pas. Mais ces derniers peuvent alors être pris en compte par un pas plus petit, si ces derniers se trouvent en bordure. Le phénomène peut être mieux compris en lisant les schémas suivants.

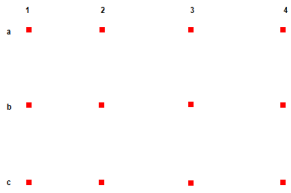


FIGURE 15: En écrasant la dimension Z, on a ce nuage de point avec un repère pour mieux comprendre sa disposition en trois dimensions.

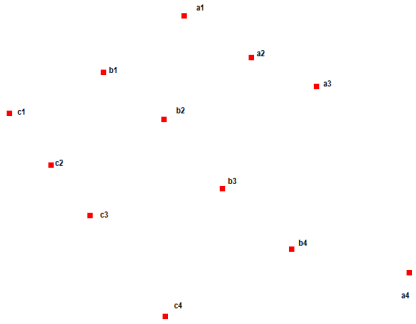


FIGURE 16: Le même nuage de point en trois dimensions. On peut alors noter que le point a4 est particulièrement en retrait, en supposant qu'il en est de même pour l'hypothétique point a5.

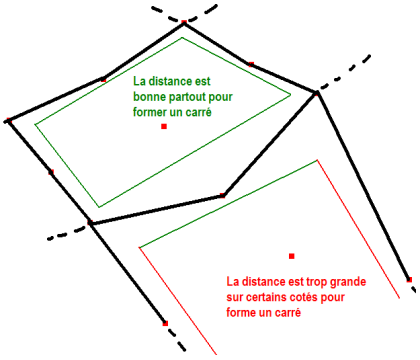


FIGURE 17: Le premier découpage se fait avec un pas de deux. On constate alors qu'il est possible de former un carré a1-a3-c3-c1.

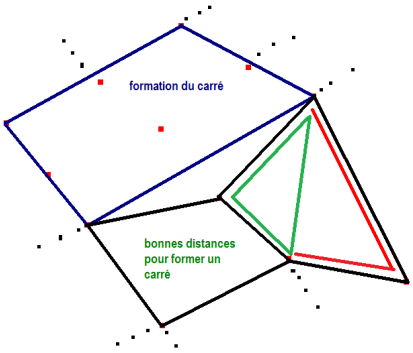


FIGURE 18: Le second découpage s'effectue alors avec un pas de un. On constate également la possibilité de former un carré b3-b4-c4-c3 et un triangle a3-b4-b3

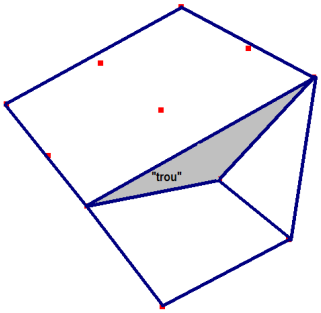


FIGURE 19: Un espace apparait alors entre les carrés formés. C'est une couture entre polygones.

Trois idées ont été trouvées pour résoudre ce problème :

1. La première consiste à réaliser un algorithme qui détecte les trous, et qui crée des polygones de la forme des trous pour les combler. Trop couteux en temps de calcul.
2. La seconde consiste à changer la façon de découper les carrés soumis à un pas plus grand que le pas minimum, en prenant en compte les points intermédiaires. En écrasant la dimension Z, ils seraient des carrés, mais en trois dimensions, ils seraient des formes quelconques. Là aussi trop couteux en temps de calcul.
3. La troisième consiste à «jouer au même jeu». Puisqu'il y a interpolation linéaire dans la formation du carré, il suffit d'interpoler linéairement les points intermédiaires. C'est une transformation à la volée du nuage de points. Les schéma suivants en expliquent le fonctionnement :

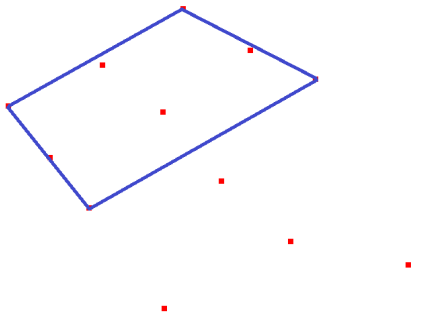


FIGURE 20: Après la premiere itération, un carré est formé

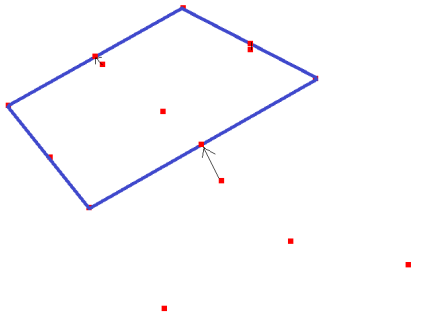


FIGURE 21: Avant de passer à la seconde itération, une interpolation linéaire est effectuée pour les points intermédiaires situés sur les cotés du carré formé

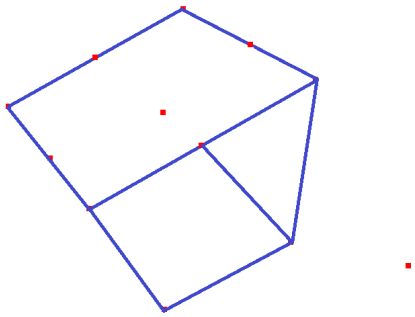


FIGURE 22: A la dernière itération, le mesh est construit et il n'y a plus de coutures entre polygones.

3.2 Conception

Ce qui a été ajouté au projet de base lors de ce stage peut être spécifié avec le diagramme état/transition suivant :

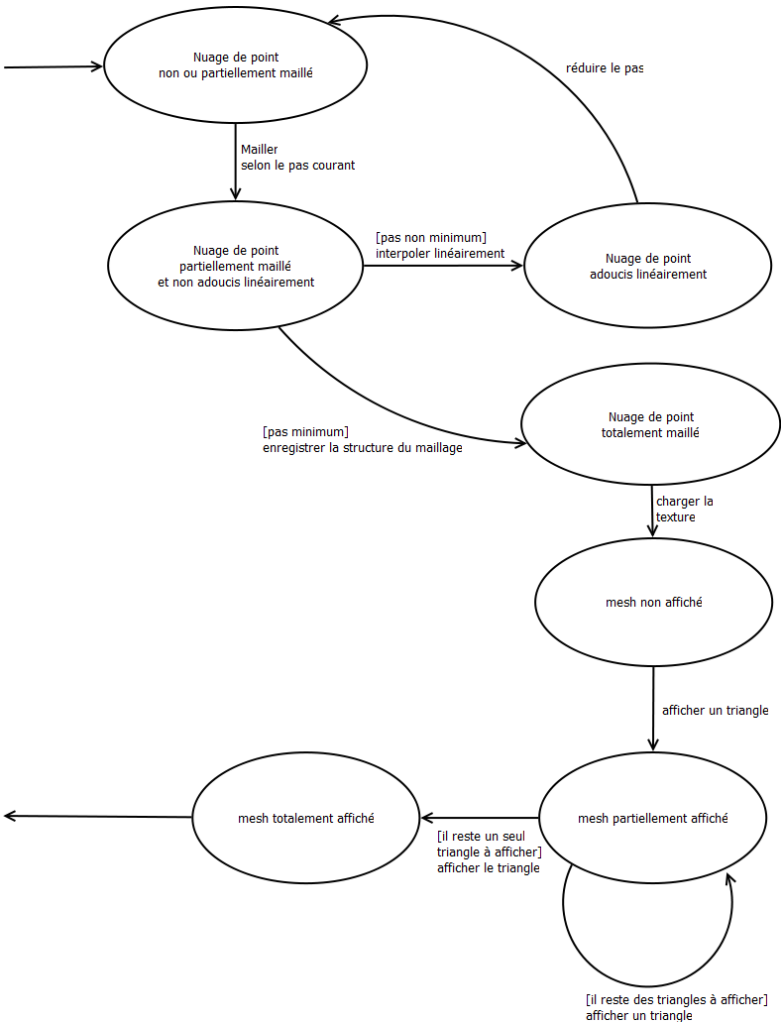


FIGURE 23: Diagramme état/transition décrivant le comportement de l'appendice de maillage du projet

3.3 Implémentation

Le programme Kinect Viewer Light de M. Penelle est développé en c++ avec le *framework* QT. L'environnement pour se stage était un pc muni d'un processeur dual core 2,7 ghz, 3Mo de ram, une carte graphique NVidia Geforce 8500 GT 1,5go et un système d'exploitation Windows 7 pro 64 bits.

Il fallait donc configurer correctement l'environnement de travail pour pouvoir compiler le programme et travailler dessus. Il a donc été nécessaire d'installer les librairies

- OpenNI (NI pour *Natural Interfaces*) qui sert dans la recherche de M. Penelle pour l'analyse de mouvements naturels.
- OpenCV (CV pour *Computer Vision*) qui est utile au traitement de l'image
- Freeglut qui garantit la portabilité de l'OpenGL. OpenGL est une API multiplateforme qui permet la génération d'images 2D et 3D.

Le compilateur utilisé est msvc2008.

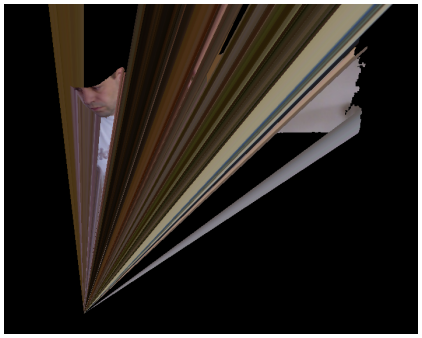
Concernant la prise en main du programme déjà réalisé, l'affichage du nuage de point se faisait dans une classe héritant de GlWidget, et dont deux méthodes doivent être citées car ce sont elles qui auront été modifiées au cours de ce projet :

- *void MyGLWidget : : refresh(Cloud *cloud)*
  Cette méthode est appelée à chaque fois que le nuage de point est modifié. C'est dans cette méthode qu'il faut calculer le mesh associé au nuage de point.
- *void MyGLWidget : : paintGL()*
  Cette méthode est appelée à chaque fois que l'univers 3D doit être affiché, c'est-à-dire à chaque frame, il faut inclure dedans du code qui se contente d'afficher ce qui a déjà été calculé précédemment.

L'affichage du nuage de points utilisait la fonction OpenGL *glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid * indices)* qui prend notamment en paramètre le type de formes à afficher ainsi qu'un tableau d'indirection. Ce procédé permet de calculer rapidement les éléments à afficher. Ici, il s'agit de points, le tableau renseigne alors simplement l'ordre dans lequel les afficher. Mais s'il s'agissait de triangles, le tableau indiquerait alors trois-par-trois les points constituant chaque triangle à afficher.

La première approche a été d'exploiter cette fonction. Le premier algorithme remplissait le tableau de triplets de points constituant systématiquement les sommets d'un triangle. Le problème de cette fonction est de ne pas pouvoir admettre de vide dans l'affichage. Or, il y a des endroits que le Kinect omet d'enregistrer, tout comme certaines zones ne peuvent être reliées. Ces points non-existant ont pour coordonnées (0,0,0) dans le nuage de point. Le premier rendu reliait alors les contours à l'origine.

Pour s'adapter à ce fonctionnement, une idée a été de relier les contours, afin de vérifier si l'algorithme fonctionnait au moins correctement. Aucune zone n'était alors inutilisée mais le maillage était réalisé sur l'intégralité de la surface.



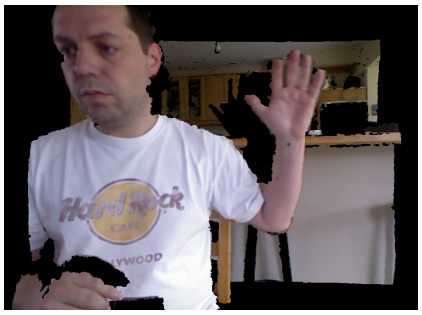
(a) Maillage en prenant en compte tous les points

(b) Maillage en prenant en compte les points non nuls

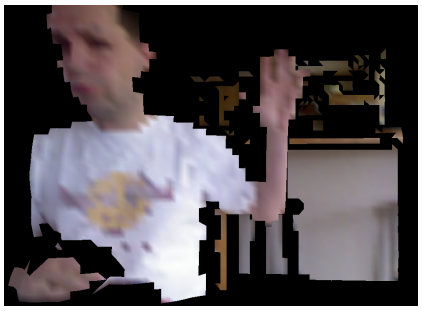
FIGURE 24: Premiers tests de maillage avec la fonction GlDrawElements

Solution retenue :
À défaut de trouver une fonctionnalité OpenGL permettant de construire le mesh à partir d'un tableau d'indirection et acceptant les vides (ce que l'on verra dans la section consacrée aux alternatives avec le GLSL), il a fallu coder une fonction qui affiche un triangle à partir de trois points, et faire appel à cette fonction pour tous les points du tableau.

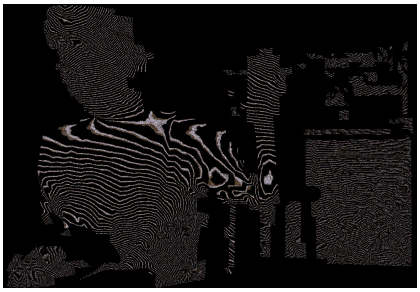
L'affichage ayant une précision au point près, il n'était pas nécessaire de lui appliquer une texture, car chaque triangle était constitué de trois points, chacun étant adjacents et constituant eux-mêmes des *textels*.
Mais la première optimisation consistant à augmenter le pas afin de construire le mesh posait alors le problème de la texture. En effet, en ne choisissant alors qu'un point sur N, on perdait alors l'information de N-1 textels. Chaque triangle étant colorié selon la méthode de *Gouraud*, on se retrouvait avec une texture floue.
Il fallait donc exploiter l'image des couleurs afin d'en faire une texture et l'appliquer sur le mesh.



(a) Maillage avec un pas de 1



(b) Maillage avec un pas supérieur à 1



(c) Premier essai d'application de texture

FIGURE 25: Premiers pas vers l'optimisation et l'application de texture

La solution en OpenGL consiste à utiliser des *coordonnées de texture*. Dans le principe, chaque *vertex* est associé à un *textel*, et le polygone concerné se voit alors appliqué dessus le fragment de texture contenu entre les textels. Pour plus de détails, se référer à la partie Documentation.

FIGURE 26: Application de la texture via les coordonnées de texture à un mesh "taillé à la hache".

Il est important de noter que, contrairement à d'autres programmes 3D, celui-ci génère une texture presque à chaque frame. Il faut donc impérativement supprimer la texture via une fonction OpenGL après l'avoir utilisée sous peine d'un débordement de mémoire.

Concernant l'optimisation, afin de faciliter l'implémentation du découpage sur plusieurs pas, l'idée était d'entourer d'une couleur les lignes de découpage. Ainsi, quand la première itération avait été codée, en superposant le mesh au nuage de point, il était possible de vérifier le bon déroulement du découpage. Quand la boucle d'appel au découpage avec tous les pas inférieurs avait été créée, on pouvait contrôler le découpage aux bordures.

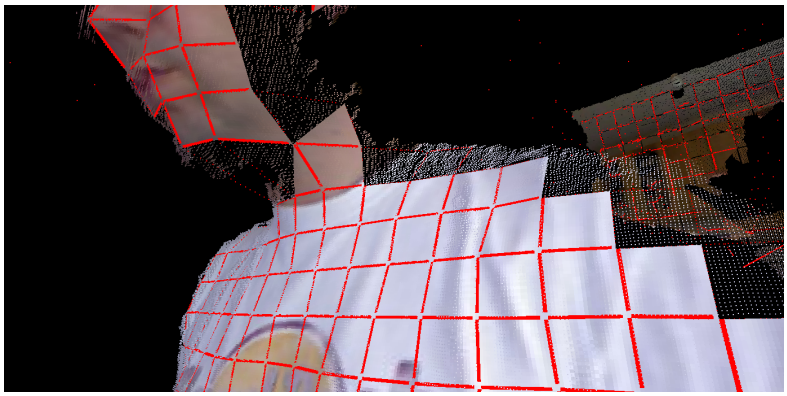


FIGURE 27: On peut remarquer l'importance de ce genre de contrôle pour constater ici que l'algorithme n'était pas encore fonctionnel concernant le maillage de tous les carrés possibles du premier pas.

Il restait à implémenter la solution retenue pour combler les vides qui apparaissaient alors. Le travail en direct sur le nuage de point semblait la solution la moins coûteuse en temps de calcul. La recopie du nuage aurait été trop lourde à effectuer à chaque frame, et est trop importante par rapport au coût de la perte de l'intégrité du nuage de point original. La modification qui lui est faite n'entraîne pas une baisse d'information significative, d'autant que le travail sur les points d'intérêt se concentre justement sur les détails qui sont ainsi générés.

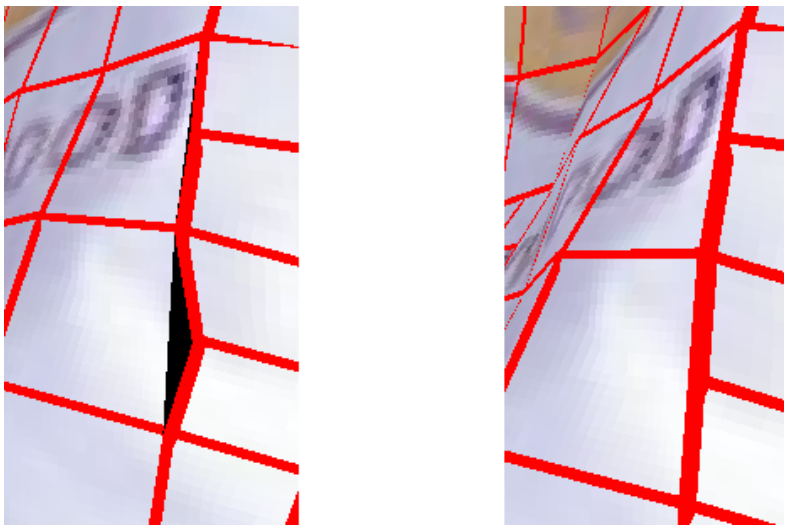


FIGURE 28: Toujours grâce aux contours, on peut alors se rendre compte des modifications apportées par l'interpolation.

3.4 Tests de performances

Le test le plus important concerne les performances. On laisse le choix du pas à effectuer, ainsi que du pas minimal sur lequel il faut s'arrêter de découper. La réduction du pas à chaque itération se fait en divisant par N, que l'on peut fixer. Voici un récapitulatif de la moyenne des FPS durant 10 secondes pour différents réglages du pas minimal et maximal avec une division du pas par 2, et en choisissant par conséquent des puissances de deux afin d'éviter des tests inutiles, du fait de la division entière en C.

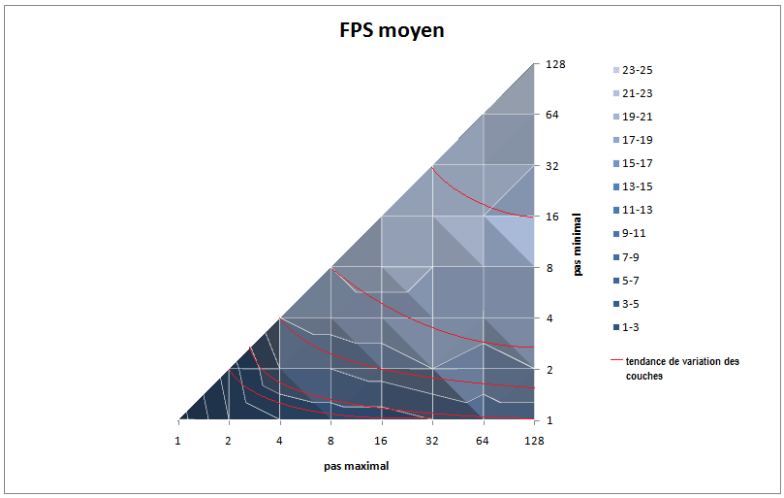


FIGURE 29: Résultats des tests en terme de FPS moyen pour plusieurs découpages.

Ce qu'il est important de constater sur ce graphique est la tendance de variation de la moyenne des FPS. On remarquera alors que choisir un gros découpage sans détails apporte un gain en FPS, ce qui est normal. Choisir un niveau de détail fin entraîne une chute de FPS. Mais plus le pas maximal est grand, et moins le pas minimal petit entraîne de perte de FPS.
Ceci est plus évident quand on écrase une dimension en réduisant le résultat de la dimension à la moyenne des valeurs qu'elle contenait :

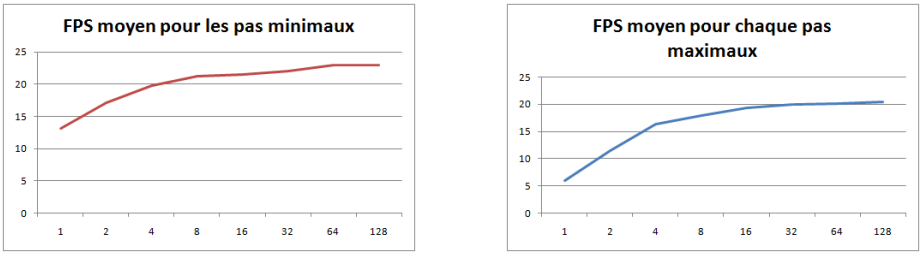


FIGURE 30: Evolution des moyennes de moyennes de FPS pour les deux pas paramétrables

Comment décider ?
Pour avoir un bon *framerate* tout en conservant un niveau de détail assez fin, il faut fixer un découpage maximal grand. Fixer un découpage minimal grand n'entraîne pas une aussi forte augmentation en FPS, donc choisir d'avoir du détail sur les contours est moins punitif en terme de performances.
Par exemple : d'après les deux graphiques, on peut dire qu'un pas maximal de 32 et un pas minimal de 4 assurent un framerate en moyenne supérieur à 20.
Il est à noter une asymptote à 24 FPS. Le programme sans algorithme de maillage et avec l'affichage uniquement du nuage de point tourne au maximum à 24 FPS sur cette machine. Il est donc logique de ne pas pouvoir les dépasser avec l'algorithme de maillage en plus.

3.5 Alternative

Ce qu'on peut constater après avoir fait tourner le programme, c'est l'usage intensif du CPU.

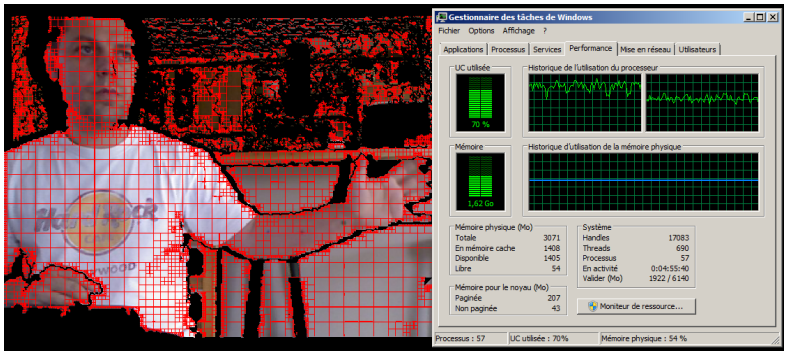


FIGURE 31: Utilisation du CPU.

Une alternative qui a été envisagée mais dont le temps à manqué pour être concrétisée est d'utiliser le GPU pour réaliser le maillage, en plus de l'affichage. La solution repose sur une idée trouvée par un Andrew Maimone, un étudiant

de l'Université de Caroline du Nord [1] :
Construire un *mesh template*, c'est à dire une surface maillée de triangles. Appliquer un *vertex shader* (exécuté coté GPU) sur le mesh afin de modifier les attributs de ses *vertices* que sont leurs coordonnées pour la forme et les couleurs pour la texture. Enfin, appliquer un *geometric shader* (exécuté coté GPU) sur le mesh afin de lui faire subir une extrusion, pour supprimer les polygones qui relient les points trop éloignés pour être reliés.

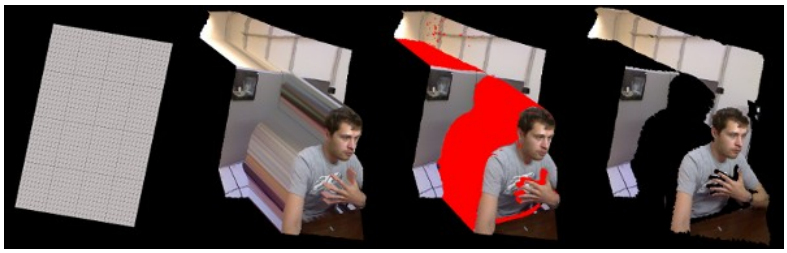


FIGURE 32: La technique employant le GPU implémentée par Andrew Maimone

Elle présente cependant un défaut : celui d'épouser moins bien la forme (cf : figure 12)

Pour son implémentation, la technique aurait nécessité d'utiliser le langage GLSL. Ce langage apporte des types comme des vecteurs ou des matrices et permet d'effectuer simplement des opérations dessus. Le code écrit est alors placé dans des fichiers propres au GLSL qui sont appelés dans le programme principal sous forme de variables de type *program*. Ces derniers sont alors compilés et font intervenir le GPU. Il s'agit de programmation sur le pipeline graphique.
Le temps a manqué pour prendre en main la technique et l'implémenter. Toutefois, il existe une ressource intéressante et en français afin de découvrir le GLSL sous QT :
`http://gbelz.developpez.com/remi-achard/gpu-avance-avec-qt/`

Une seconde façon d'approcher cette alternative était l'utilisation de CUDA. Il s'agit de GPGPU (*General-Purpose Computing on Graphics Processing Units*), c'est à dire qu'avec cette technique, il est possible de faire réaliser par le GPU des calculs qui sont habituellement réalisés par le CPU. Le langage utilisé peut être alors du C avec des fonctions issues de la librairie CUDA. L'idée a été émise en fin de projet donc aucune recherche n'a pu être effectuée à ce sujet, exception faite de la constatation suivante : cette idée n'est réalisable qu'avec les cartes graphiques compatibles CUDA (à partir de Geforce 8), puisqu'il est nécessaire d'avoir une architecture particulière.
Il est cependant intéressant de noter que la technologie CUDA est récente et que des tutoriels en français en expliquent le fonctionnement, comme celui de développez.com :
`http://tcuvelier.developpez.com/tutoriels/gpgpu/cuda/introduction/`

4 Documentation technique

Cette partie a pour but de détailler le code et les fonctions utilisées pour implémenter la technique de maillage. Pour du détail ligne par ligne, se référer directement au code source totalement commenté.

4.1 Algorithme de maillage

L'algorithme se base sur de nombreuses fonctions bas niveau. Sa signature est la suivante :

```
int MyGLWidget::maillage(int flou_resolution_3d,
 int debutx,
 int debuty,
 int finx,
 int finy,
 int k)
```

Et son code, résumé en pseudo-langage universel afin de ne pas recopier un listing en C qui serait bien trop grand à publier dans ce document :

```
Soit un quadrilatère de coin supérieur gauche
de coordonnées (debutx,debuty)
et de coin inférieur droit
de coordonnées (finx,finy)
et quadrillé selon un pas
de longueur flou_resolution_3d

mode = maillage

tant que mode =/= fin

  Pour chaque case
    Si les coins sont rapprochés sur la dimension Z
      ET mode = maillage
        les ajouter au tableau d'indirection aux cases
        k, k+3, k+6 (et incrémenter k de 9),
        appeler la fonction d'interpolation
        sur les cotés concernés.
    Sinon, Si mode = detail
      Appliquer la fonction maillage sur cette case
  avec flou_resolution_3d plus petit
  Fin pour

  si mode = maillage alors mode = detail
  si mode = detail alors mode = fin

fin tant que

condition d'arrêt : flou_resolution_3d est
inférieur ou égal au pas minimal
```

La question concernant k : pourquoi récupérer l'indice du tableau alors qu'il existe de nombreuses méthodes pour traiter les tableaux en c++ ?
La réponse est dans les performances. Premièrement, il n'est pas question de réaliser des réallocations de mémoire à chaque tour de boucle, c'est trop couteux en mémoire. Il existe plusieurs classes en c++ (dont Vector) qui font office de tableau et dont on peut spécifier la taille dès le début, mais on perd l'efficacité en espace mémoire qu'offre le pointeur de base.

Le premier appel de la fonction se fait depuis la méthode *refresh*

```
k=this->maillage(flou_resolution_3d_global,0,0,640*3,480,0);
```

Le coin supérieur gauche a pour coordonnées (0,0), l'inférieur droit (640x3,480) puisqu'on travaille sur 640x3 données par lignes, pour 640 triplets de coordonnées par ligne.
La variable donnant le pas maximal (flou_resolution_3d_global) est un attribut de la classe myglwidget et peut donc être modulé à volonté.

L'algorithme utiliser une fonction d'interpolation dont voici la signature

```
 void MyGLWidget::interpollation
(int debut, int fin, int pas, int flou_resolution_3d)
```

flou_resolution_3d est utilisé dans la formule d'interpolation, pas est le pas selon lequel on choisi les points à interpoler, pour ne pas le faire sur tous quand cela n'est pas utile.

4.2 Affichage

On se situe à présent dans la méthode *paintGL*.
On crée un tableau carré dont le nombre de case de coté est une puissance de 2 (ici, 1024). On entre dedans les informations de l'image de couleurs (640x480). Ce tableau s'appelle Texture3.
La suite est directement commentée. Il s'agit de créer la texture OpenGL et la charger, en utilisant Texture3.

```
//nous avons une texture de 1024x1024 textels codés
//avec 3 ubytes
//activons tout d'abord le texturing
  glEnable(GL_TEXTURE_2D);
//ensuite on cree une texture vierge
  GLuint Nom;
  glGenTextures(1,&Nom);  //Génère un n° de texture
  glBindTexture(GL_TEXTURE_2D,Nom);  //Sélectionne ce n°
//on affecte notre texture
  glTexImage2D (
           GL_TEXTURE_2D,  //Type : texture 2D
           0,  //Mipmap : aucun
           GL_RGB,  //Nombre de couleurs (3)
           1024,  //Largeur
           1024,  //Hauteur
           0,  //Largeur du bord : 0
           GL_RGB,  //Format : RGB
           GL_UNSIGNED_BYTE,  //Type des couleurs
           Texture3 //Addresse de l'image
           );
//parametrage du rendu
//gestion des cas ou le pixel est plus grand que les textels :
// GL_NEAREST = on lui applique la valeur du textel le plus
// proche (distance de manhattan) de son centre
//NB : pour un filtre moyenneur, utiliser GL_LINEAR, qui
// se base sur des groupes de 4 textels les plus proches.
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
  GL_NEAREST);
//gestion des cas ou le pixel est plus petit que les textels :
//GL_NEAREST = on lui applique la valeur du textel
//le plus proche (distance de manhattan)
// de son centre. Idem pour le filtre moyenneur
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
  GL_NEAREST);
//gestion des cas ou l'on arrive en bord de texture :
//Répéter la texture à l'infini.
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
  GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
  GL_REPEAT);
//concernant le plaquage de la texture :
//GL_NICEST pour un plaquage exact de la texture
//GL_FASTEST pour une approximation (gain en FPS car
// c'est une simple interpolation linéaire)
  glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_FASTEST);
//affectation de notre texture avant application
  glBindTexture(GL_TEXTURE_2D,Nom);
```

Enfin, on utilise une structure Point :

```
Point(float x, float y, float z, quint8 r,
quint8 g, quint8 b, int coord)
```

(x,y,z) sont ses coordonnées dans l'espace. (r,g,b) sont ses composants de couleur codées sur 8 bits. coord est l'emplacement du point dans l'image des profondeurs et/ou l'image des couleurs.

On construit 3 points en utilisant trois points du tableau d'indirection, et on dessine le triangle associé dans l'espace via la fonction Dessiner_triangle

```
void MyGLWidget::Dessiner_triangle
(point a, point b, point c, quint8* rgb)
```

Cette fonction a du code commenté, parce qu'elle peut faire double-usage. Soit utiliser la texture précédemment créée soit utiliser l'image des couleurs si l'on souhaite ne pas utiliser de texture openGL. (Il faut alors utiliser le paramètre rgb) La fonction se sert de la fonction OpenGL *glBegin(GL_TRIANGLES);*

5 Déroulement du stage

5.1 Gestion du projet

Le sujet avait été communiqué plusieurs mois à l'avance, et j'ai ainsi pu m'organiser et prendre un peu d'avance sur les connaissances théoriques de la 3D en temps réel en proposant mon propre projet multimédia à l'École : Un travail de recherche sur L'évolution de la 3D temps réel.

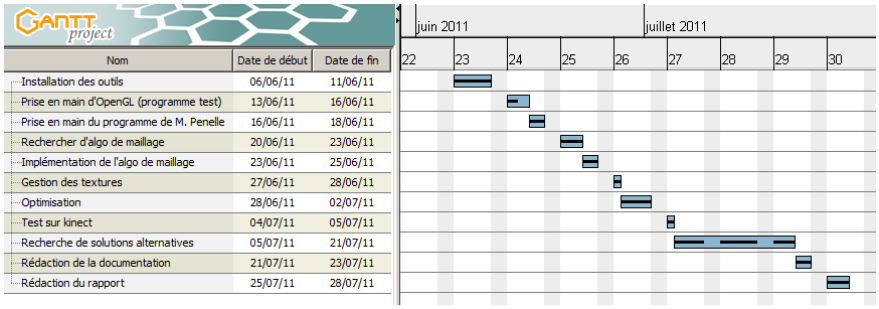
Connaissant alors les termes techniques et une partie des enjeux, j'ai pu faire avec moins de difficultés une approche du projet après la réunion de lancement.

M. Penelle m'a aidé à installer toutes les librairies et configurer l'environnement de travail et m'a orienté dans la recherche de solutions.

Chaque semaine, une réunion d'avancement avait lieu en présence des autres stagiaires. A la moitié du stage, j'ai rendu une planification effective et prévisionnelle, et le premier prototype du programme a pu être implémenté sur la machine de M. Penelle.

A la fin du stage j'ai rendu ce rapport avec la documentation technique à M. Penelle, ainsi que le programme terminé.

5.2 Planification



Nom	Date de début	Date de fin
Installation des outils	06/06/11	11/06/11
Prise en main d'OpenGL (programme test)	13/06/11	16/06/11
Prise en main du programme de M. Penelle	16/06/11	18/06/11
Rechercher d'algo de maillage	20/06/11	23/06/11
Implémentation de l'algo de maillage	23/06/11	25/06/11
Gestion des textures	27/06/11	28/06/11
Optimisation	28/06/11	02/07/11
Test sur kinect	04/07/11	05/07/11
Recherche de solutions alternatives	05/07/11	21/07/11
Rédaction de la documentation	21/07/11	23/07/11
Rédaction du rapport	25/07/11	28/07/11

FIGURE 33: Planification effective du stage

6 Conclusion

Le maillage en temps réel de nuages de points est un sujet vaste qui peut mettre en jeu de nombreuses solutions. L'évolution des cartes graphiques apporte continuellement de nouvelles façons de générer de la 3D en temps réel. La solution implémentée utilise une méthode qu'on considère comme "dépréciée" de nos jours, car elle utilise en trop grande partie le CPU. Mais l'optimisation et la puissance des machines ont permis d'obtenir un résultat satisfaisant. Avec une durée plus longue et une formation préalable sur le calcul sur GPU, il est possible de créer une solution qui exploiterait au maximum la carte graphique.

Ce stage aura été pour moi une introduction intéressante et une vrai application dans le monde de la 3D en temps réel, et je suis particulièrement intéressé pour approfondir mes connaissances sur ce sujet passionnant, au vu des enjeux que soulèvent les constructeurs de cartes graphiques.

7 Bibliographie

Références

[1] Andrew Maimone, étudiant de l'Université de Caroline du Nord, *Encumbrance-free Telepresence System with Real-time 3D Capture and Display using Commodity Depth Cameras*, `http://www.cs.unc.edu/~maimone/KinectPaper/kinect.html`, page consultée le 26 Juillet 2011.

[2] Bérénice, *La manette c'est vous !*, freak-geek.com, 15 novembre 2010, `http://www.freak-geek.com/la-manette-cest-vous/`, page consultée le 19 Juillet 2011.

[3] Images d'exemple de l'article Normal mapping de wikipedia, `http://fr.wikipedia.org/wiki/Normal_mapping/`, page consultée le 19 Juillet 2011.