Université du Québec en Outaouais
Département d'informatique et d'ingénierie

Baccalauréat en informatique
Projet Synthèse (*INF4173*)
Hiver 2007

Visualisation 3D d'imagerie médicale

Rapport final

Étudiant
Jacquelin Caron

Superviseur
M. *Jean-François Lapointe*, M.Ing., Ph.D.

Coordinateur
M. *Michal Iglewski*, Ph. D.

1. Table des matières

2. Introduction

Le but principal de ce projet est de pouvoir offrir un point de vue en 3D d'images médicales. Puisqu'il n'est pas facile pour certaine personne de faire la représentation 3D dans sa tête, un outil comme le notre permettra donc de créer cette version en 3D.

Afin de mettre l'accent sur différente partie à analyser, il est possible d'éliminer certaine information pour plus facilement en afficher d'autre.

Pour des raisons de portabilité, les données en 3D sont représentées à l'aide du langage VRML puisque celui-ci est disponible, à l'aide de visionneur, sur plusieurs plateforme (*Windows, Linux, Unix, …*).

3. Objectif

Ce projet a donc pour but d'offrir une représentation 3D du contenu de fichier d'imagerie médicale (*fichier DICOM*) à l'aide de différents procédés et technologies développées dans les dernières années. Le projet sera divisé en quatre grands volets :

1. Convertisseur de DICOM vers des données brutes
2. Transformation des données brutes en données 3D VRML
3. Manipulation des images 3D
4. Amélioration des données 3D
    I. Réduction de la taille des modèles 3D
    II. Coloration des voxels

4. Calendrier

Date	Tâche
Avant 16 janvier	Analyse du format DICOM
16 janvier	Rencontre d'information
25 janvier	Remise du plan de travail
16 janvier - 26 janvier	Convertisseur de DICOM vers données brutes
29 janvier – 9 mars	Transformation des données brutes en VRML
5 mars	Remise du rapport de progrès
12 mars – 2 avril	Amélioration des données 3D
4 avril	Présentation
20 avril	Dépôt du rapport final

5. Division du travail

Pour réaliser ce projet, il a fallu dès le début séparer celui-ci en deux parties. Ces deux parties simplifient la complexité du problème et permet surtout de limite le nombre de conversion à effectuer.

5.1. DICOM vers les données brutes

Cette première partie, nommée « *dicom2raw* », consiste donc à récupérer les images dans le fichier, ou les fichiers, DICOM et de les convertir dans un format de fichier où il y a seulement les informations suivantes :

1. Largeur d'une image
2. Hauteur d'une image
3. Le nombre d'image
4. La distance entre les tranches d'image (*où* 1 *représente la dimension d'un seul pixel*)
5. La couleur (*en échelle de gris*) de chaque pixel de l'ensemble d'image

5.2. Les données brutes vers VRML

La deuxième partie du projet, nommée « *raw2vrml* », utilise en entrée la sortie de la première partie et produit un fichier VRML. La production du fichier VRML est fait en mettant l'accent sur une zone d'acceptation selon la gamme de couleur (*en échelle de gris*).

Donc, les **voxels** (*pixel 3D*), contenant des parties se trouvant à l'extérieur et à l'intérieur de la limite, sont convertis, à l'aide de l'algorithme des « *marching cubes* », en des données de la surface de l'objet en 3D.

6. Les images médicales

Les images médicales sont de format DICOM. C'est le format d'image standard pour la plus part des « *scanners* » utilisés dans le milieu médical.

Puisque le format DICOM est très large, il y a eu certaines parties qui ont été négligée afin de pouvoir mettre l'accent sur des résultats. Donc, pour le moment, les DICOM qui utilisent une compression d'image (*comme JPEG ou Run-Length*) ne sont pas gérés pour le moment. Les images utilisées pour faire les tests sont donc non compressées.

De plus, les images utilisées proviennent de deux sources :

1. Internet, avec quelques sites qui offrent des images typiques
2. La clinique **IRM St-Joseph** qui nous ont fourni des cas réels et concrets.

6.1. L'analyse des images

L'analyse du format DICOM a été fait en grosse partie à l'aide de la documentation disponible sur le site officiel du format (*http://medical.nema.org/*). La description du format DICOM est regroupée dans 18 documents. Mais seulement les parties suivantes ont été nécessaires pour le côté informatique :

6.1.1. Partie 5 : Structure de données et Encryptions

Cette partie contient les informations sur la disposition des champs de donnée dans l'entête du fichier ainsi que la nature de chaque donnée. Cette partie conclue avec la technique à utiliser pour extraire les données de chaque pixel.

6.1.2. Partie 6 : Dictionnaire de données

Ici, il y a la liste complète des champs situés dans l'entête du fichier DICOM. Chaque champ est identifié à l'aide de deux valeurs de deux octets :

1. L'identificateur de groupe
2. L'identificateur du champ

L'analyse de ces champs a été des plus vital pour le projet puisque c'était nécessaire pour extraire les informations des fichiers DICOM. La catégorie de champs la plus utilisée est la 0028h. Ces champs correspondent aux informations sur les images dont :

1. Le nombre de ligne
2. Le nombre de colonne
3. Le nombre d'image
4. L'interprétation photométrique (*la façon dont les données, principalement sur chaque pixel, sont emmagasinées*)
5. Les bits d'allocation pour chaque pixel
6. Le bit supérieur pour chaque pixel

6.2. Regroupement d'images

Une autre constatation faite durant l'analyse du format DICOM, c'est qu'il y a deux façons de garder chaque tranche d'image :

1. L'ensemble des tranches dans le même fichier
2. Une image par fichier

Sur ce fait, il a donc fallu adapter la première partie du projet afin de pouvoir faire face à ce fait. Et dans les deux cas, il y a la création d'un seul fichier de sortie qui contient l'ensemble des tranches d'image dans leur format brute.

7. Le langage VRML

Pour les besoins de ce projet, il y a donc eu l'apprentissage des fonctionnalités de base du langage VRML. Le VRML permet de modéliser les environnements en 3D. C'est donc l'outil idéal pour reproduire en 3D les images médicales.

Il y a déjà eu deux versions pour l'utilisation du VRML. Une première, plus expérimentale, utilisait la notion de « *prototype* ». Cette solution, qui a fonctionné rapidement, était

énormément exigeante sur la machine et n'étaient pas une solution optimale pour VRML puisque les prototypes n'étaient pas conçus pour être appeler autant de fois.

Par la suite, il y a donc eu la création de la deuxième solution, qui permet de considérer une énorme quantité de donner et sans aucun problème. À l'intérieur de cette solution, il y a la création d'un tableau avec les coordonnées utilisées. Et puisque chaque coordonnée est utilisée quatre fois, ceci diminue le nombre de calcul à faire, par rapport à la première, énormément. De plus, cette seconde version utilise des propriétés plus communes du VRML. Donc, des fichiers VRML de *quatre-vingt* mégaoctets peuvent facilement être géré par les visionneurs VRML.

De plus, avec le VRML, ce qui est intéressant, c'est que la manipulation des modèles 3D est intégrée dans les différents visionneurs. Il suffit donc d'indiquer dans le fichier VRML quelle manipulation on désire utiliser.

Lors de la présentation du projet, le visionneur VRML utilisé est Démotride (*http://www.demotride.net/intro.html*).

8. Les « marching cubes »

L'algorithme des « *marching cubes* » est la version tridimensionnelle de l'algorithme des « *marching squares* ». Cet algorithme permet donc de créer une surface en examinant chaque voxel. Tout dépendant des huit pixels qui composent le voxel, où chaque sommet du cube est donc un pixel, il y a des décompositions valides afin de créer une surface correspondant. Chaque pixel peut être considéré à l'intérieur ou à l'extérieur d'une limite, et c'est à partir de cette évaluation qu'il est possible de choisir quelle décomposition va déterminer la surface.

Il y a, en considérant les rotations et négation des sommets internes et externes, quinze possibilités de décomposition pour les « *marching cubes* ». Voici donc ces quinze décompositions (*après les opérations de négation et de rotations*) :

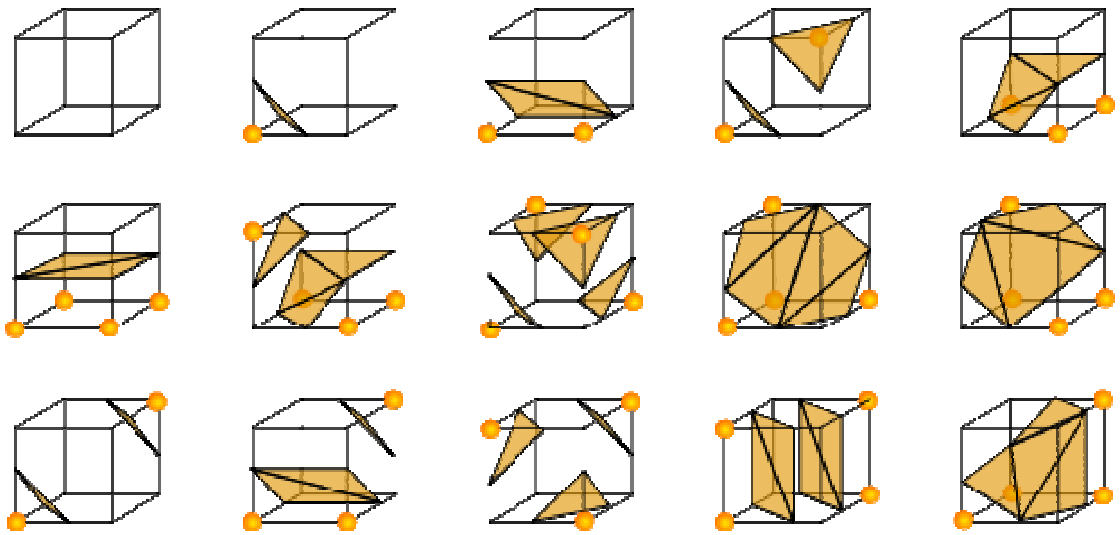

Figure 1

Dans cette figure, les sommets en jaunes représentent ceux qui sont à l'intérieur de la surface et les autres, à l'extérieur.

Dans le cadre de ce projet, il y a eu la création d'un algorithme pour générer de façon dynamique les 256 décompositions possibles. La création de cet algorithme a été nécessaire puisque la documentation sur Internet est assez limitée. Une fois les 256 « *marching cubes* » générés, il est possible d'utiliser la décomposition faite pour chaque voxel examiné.

8.1. Les concepts derrière l'algorithme développé

Comme on peut l'observer sur la **figure 1**, les formes sont créées en reliant ensemble des points au milieu des arrêtes du cube. Mais il y a une condition très importante pour qu'un de ces points soit considéré pour faire partie d'une des formes : il faut que les deux extrémités de l'arrête sur laquelle il est ne soient pas de même côté de la surface à construire.

Par la suite, pour donner une apparence tridimensionnelle à ces formes, il suffit de les trianguler. Puisque celles-ci sont toutes convexes, il est très facile de faire une triangulation et il n'y a pas à faire face aux problèmes que des formes concaves peuvent causer. On a donc un algorithme d'ordre linéaire pour créer les triangles nécessaires.

Pour chaque cube, après la triangulation, il y a un maximum de quatre triangles. Ceci fourni une information des plus importante qui est la borne supérieure sur le nombre de triangles possibles. Naturellement, la borne inférieure est le cas où il n'y a tout simplement pas de triangle.

C'est donc en se basant sur ces deux bornes qu'il est possible de déterminer l'espace mémoire nécessaire pour contenir l'information sur chaque cube généré.

8.2. L'algorithme développé

L'algorithme développé pour les besoins du projet peut se séparer en trois parties :

1. Le corps
2. La recherche en profondeur
3. Le tri en largeur
4. La création de triangle
5. L'utilisation des « *marching cubes* »

Les règles de syntaxe pour les algorithmes sont expliquées en annexe.

8.2.1. Structure de donnée du cube et ses fonctions

- Les points du cube $\mathcal{V}$, où v $\in \mathcal{V}$
    - *side*[v] : fonction qui informe de quel côté le point se situe par rapport à la surface
- Les points au centre des arrêtes $\mathcal{M}$, où m $\in \mathcal{M}$

- *active*[m] : informe si le point est activé ou non. Activé signifie qu'il sera éventuellement considéré pour créer les triangles pour la surface. Au commencement de l'algorithme, tous les points sont considérés comme étant activés.

- Les arêtes du cube $\mathcal{E}$, où $\mathcal{E} = \{(u, v, p) \mid u, v \in \mathcal{V} \wedge p \in \mathcal{M}\}$

  - *adj*[v, $\mathcal{E}$] : retourne l'ensemble des points adjacent au point **v, v** $\in \mathcal{V}$. Il est nécessaire d'avoir l'ensemble des arêtes $\mathcal{E}$ puisque c'est cette structure qui contient les liens.
  - *mid*[v, $\mathcal{E}$] : retourne l'ensemble des points au milieu des arêtes qui sont liés au point **v, v** $\in \mathcal{V}$. Seul les points actifs sont retournés. Tout comme dans la fonction adj, l'ensemble $\mathcal{E}$ est présent puisqu'il contient la structure qui représente les liens.

- Le graphe qui représente le cube est G, G = (**V**, **E**) | V = $\mathcal{V}$ ^ E = $\mathcal{E}$

8.2.2.Le corps

C'est la fonction d'entrée dans l'algorithme. Celle-ci est appelée pour générer la triangulation d'un cube en particulier.

La fonction *marchingCube* prend en paramètre **G** qui est un graphe décrivant le cube et retourne un ensemble de triangle qui le décompose.

```
def marchingCube(G):
  foreach e in E[G]:
    if side[u[e]] = side[v[e]]:
      active[p[e]] ← INACTIVE

  T ← {}
  foreach v in V[G] where side[v] = INTERIEUR:
    P ← depthSearch(v, E[G])
    P ← breadthSort(P, E[G])
    foreach i in range(1, |P|-2):
      T ← T ∨ createTriangle(Pi, Pi+1, Pi+2)

  return T
```

À la première boucle, il y a la vérification pour chaque arête du cube et l'inactivation des points au centre des arêtes qui ne seront plus considérer.

Par la suite, il y a l'initialisation à vide de l'ensemble **T** qui va contenir les triangulations possibles pour ce cube.

Pendant la deuxième boucle, qui s'exécute pour chaque point qui est considéré comme étant à l'intérieur, il y a un appel pour une recherche en profondeur. Par la suite, il y a un tri pour les points au centre des arêtes choisis. L'ordre de ces points permet de créer les triangles qui sont insérés dans l'ensemble **T**.

Par la suite, c'est l'ensemble **T** qui est retourné.

8.2.3.La recherche en profondeur

Ici, la recherche en profondeur fonctionne exactement comme l'algorithme connu de tous. La seule différence, c'est qu'après avoir fait son parcours, le résultat retourné est un ensemble de point au centre des arêtes qui vont être utilisé pour construire l'un des faces de la surface.

Pour éviter de repasser par des points déjà visités, le point une fois analysé change d'état pour être considéré maintenant à l'extérieur de la surface.

La fonction *depthSearch* prend en paramètre un point **v** et l'ensemble des arêtes $\mathbb{E}$.

```
def depthSearch(v, E):
  P ← mid[v, E]
  side[v] ← EXTERIEUR

  foreach a in adj[v, E] where side[a] = INTERIEUR:
    P ← P ⋎ depthSearch(a, E)

  return P
```

Il est important de mentionner qu'une fois qu'un point change de côté, ceci affecte aussi la deuxième boucle dans le corps de l'algorithme. Ce qui permet d'éliminer des itérations de trop.

8.2.4.Le tri en largeur

Le tri en largeur est simplement un mélange entre un tri et une visite en largeur. Bref, c'est simplement l'ordre de parcours qui serait nécessaire pour visiter les points d'un polygone. Comme le démonter la **figure 2**, le point de départ était le #1 et les autres points ont été visités selon le principe de la visite en largeur.

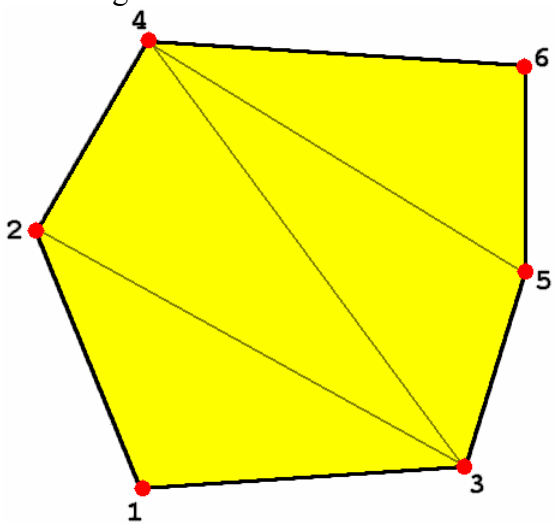

Figure 2

Donc, la fonction *breadthSort*, vu dans la première partie de l'algorithme retourne le même ensemble de point, mais cette fois-ci, trié par une visite en largeur.

La fonction *breadthSort* prend donc en paramètre **P**, l'ensemble de point milieu et **E**, la structure des arêtes pour le cube.

8.2.5.La création de triangle

Pour la création de triangle, il suffit simplement d'utiliser la version triée des points milieux fournis par la fonction précédente. Comme le démontre la **figure 2**, avec l'ordre des points, il est très facile de créer des triangles.

Pour le premier triangle, les trois premiers points sont utilisés pour le former. Par la suite, les points 2, 3 et 4 pour le second triangle, et ainsi de suite.

La fonction pour créer un triangle est `createTriangle` et prend en paramètre les trois points milieux qui seront utilisés pour le faire.

8.2.6.L'utilisation des « marching cubes »

Une fois les 256 « *marching cubes* » générés, ils sont disponibles pour être utilisable beaucoup plus rapidement.

L'astuce est d'obligatoirement parcourir chaque voxel. Chaque voxel est analysé pour déterminer à quelle décomposition des « *marching cubes* » il correspond. Par la suite, il suffit d'aller visité le bon cube et tout simplement prendre en compte chaque triangle et d'ajouter chaque point en additionnant la position du voxel, dans l'espace 3D, avec les coordonnées des points.

Cette technique est absolument nécessaire puisque chaque voxel doit être parcouru. Et le parcours de chaque voxel compose en fin de compte le « *bottleneck* » dans l'exécution. Donc, ne pas avoir à régénérer le bon cube en question, c'est une énorme économie de temps.

9. Les librairies

L'une des parties les plus importante du projet était de créer des librairies pour intégrer les fonctionnalités développées dans d'autres projets. Les deux librairies correspondent simplement aux deux parties du projet.

Les librairies sont conçues pour respecter les standards que le compilateur « *gcc* ». Donc, chaque librairie est livrée avec un fichier d'entête (**.h*) et la librairie (**.a*).

9.1. « dicom2raw »

Donc, cette librairie transfère donc les données provenant de fichiers DICOM vers un fichier de données brutes.

Le fichier d'entête : *dicom2raw.h*
La librairie : *libdicom2raw.a*

9.1.1. Détail sur les méthodes

dicom2raw1f

```
bool dicom2raw1f(FILE *out, int compress, char *in);
```

La fonction convertie un fichier DICOM, contenant l'ensemble des images nécessaires, en un fichier de données brutes.

Paramètres:

1. *out* – fichier de sortie
2. *compress* – niveau de compression pour les images
3. *in* – le nom du fichier DICOM en entré

Retour:

- Si la fonction s'exécute parfaitement, elle retourne « *true* ». Sinon, elle retourne « *false* ».

dicom2rawfv

```
bool dicom2rawfv(FILE *out, int compress, int vlen, char **ins);
```

La fonction convertie des fichiers DICOM, contenant chacun une image, en un fichier de données brutes.

Paramètres:

1. *out* – fichier de sortie
2. *compress* – niveau de compression pour les images
3. *vlen* – le nombre de fichier en entré
4. *ins* – une liste des noms de fichier en entré, de longueur *vlen*

Retour:

- Si la fonction s'exécute parfaitement, elle retourne « *true* ». Sinon, elle retourne « *false* ».

dicom2rawfa

```
bool dicom2rawfa(FILE *out, int compress, int vlen, char *i1, ...);
```

La fonction convertie un fichier DICOM, contenant l'ensemble des images nécessaires, en un fichier de données brutes.

Paramètres:

1. *out* – fichier de sortie
2. *compress* – niveau de compression pour les images
3. *vlen* – le nombre de fichier en entré
4. *i1* – le nom du premier fichier DICOM
5. … – les autres noms de fichier DICOM

Retour:

- Si la fonction s'exécute parfaitement, elle retourne « *true* ». Sinon, elle retourne « *false* ».

9.2. « raw2vrml »

Pour cette librairie, l'entrée est le fichier de données brutes généré par la première partie et il est converti en un fichier VRML.

Le fichier d'entête : *raw2vrml.h*
La librairie : *libraw2vrml.a*

9.2.1. Détail sur les méthodes

raw2vrmlInit

```
bool raw2vrmlInit(void);
```

La fonction initialise le tableau des « *marching cubes* » qui sera utilisé pour traiter les voxels.

Retour:

- Si la fonction s'exécute parfaitement, elle retourne « *true* ». Sinon, elle retourne « *false* ».

raw2vrmlDestroy

```
bool raw2vrmlDestroy(void);
```

La fonction qui libère l'espace mémoire prise par le tableau des « marching cubes ».

Retour:

- Si la fonction s'exécute parfaitement, elle retourne « *true* ». Sinon, elle retourne « *false* ».

raw2vrml

```
bool raw2vrml(FILE *out, FILE *in, float scale,
              byte min, byte max, bool acceptInLimit);
```

N.B.: typedef unsigned char byte;

La fonction convertie un fichier de données brutes en un fichier VRML selon les critères choisis.

Paramètres:

1. *out* – fichier de sortie
2. *in* – le fichier de données brutes
3. *scale* – l'étirement sur l'axe des Z
4. *min* – limite inférieure (*couleur des pixels*)
5. *max* – limite supérieure (*couleur des pixels*)
6. *acceptInLimit* – « *true* » accepte les pixels qui sont à l'intérieurs des limites, « *false* », à l'extérieur

Retour:

- Si la fonction s'exécute parfaitement, elle retourne « *true* ». Sinon, elle retourne « *false* ».

9.3. Derniers ajouts au projet

Depuis le rapport de progrès, il y a eu les ajouts suivants de fait au projet.

9.4. Compression des pixels

Puisqu'il y a énormément d'information à gérer, et que la quantité de donnée est déterminée par le nombre de voxel qu'il y a, il serait import d'offrir une option pour compresser les images. Cette compression permet de diminuer le nombre de pixel par image et donc le nombre de voxel à parcourir. Dans certain cas, il n'y a pas énormément de donnée, donc cette option permet d'accélérer le traitement sans nécessairement perdre des données (*dépendant du niveau de compression demandée*).

Une compression de **niveau 1** consisterait à prendre quatre pixels, faire la moyenne des quatre couleurs dans ces pixels et calculer la moyenne de ceux-ci. Le pixel résultant aurait donc cette nouvelle valeur. Avec une compression de **niveau 1**, une image 512x512 pixels serait converti en une image de 256x256.

9.5. Gamme de couleur

L'utilisation d'une gamme de couleur permet de mettre l'emphase sur certain détail. À l'origine, c'est seulement des tons de gris qui ont été utilisés pour représentés les images. Mais en divisant cet échelle de gris en quatre parties qui utilisent des couleurs différentes permettrait de voir plus facilement les différentes entre les voxels.

Figure 3

La **figure 3**, représente grossièrement la modification qui a été apportée. Les deux sont équivalents, mais même avec l'exemple précédent, il est facile de voir que les couleurs font plus facilement ressortir certain détail.

De plus, avec la modification des couleurs, la limite inférieure correspond au rouge le plus à gauche et la limite supérieure est associée au bleu de plus à droite. Donc, plus la limite est mince, plus il y a une différence visible entre chaque voxel.

9.6. Traduction

Pour augmenter la porter des deux logiciels, il y aura la possibilité de produire une plus facilement une traduction. Les logiciels sont donc originalement disponibles en Français et en Anglais.

De plus, les logiciels ont été conçus pour facilité la traduction en d'autre langue. Il est donc possible de rendre le logiciel encore plus convivial pour certaines personnes.

9.7. Compilation Linux

Puisque les deux logiciels fonctionnent en ligne de commande et qu'ils sont développés en **C**, il est maintenant possible de compiler les librairies, ainsi que leurs interfaces, sur Linux (*pour des processeurs compatible x86*).

10. Développements futurs

Malgré le fait que le projet soit terminé, il y a place à plusieurs autres développements possibles à apporter. Voici donc quelques ouvertures possibles.

10.1. Migration du VRML au X3D

Le langage X3D est le successeur du VRML. Avec l'utilisation du X3D, ceci permettrait de s'assurer que le projet suive l'évolution des technologies 3D.

10.2. L'interpolation

L'interpolation consiste à approximer de l'information avec celle que l'on a déjà. Pour le projet présent, ce qui sera intéressant est de créer une ou des images entre deux images que l'on a déjà à notre disposition.

Comme la précision de la surface dépend du nombre d'image que l'on possède. L'interpolation permettrait de rajouter de cette information en question. Et avec plus d'information, il serait possible de pouvoir offrir une meilleure idée des formes que l'on veut analyser.

Naturellement, en ajoutant des images, ceci étire le modèle sur l'axe des Z. Donc, avec une interpolation, il ne faudra pas oublier d'ajuster la valeur d'étirement (*scale*).

10.3. Une interface visuelle

Présentement, le projet fonctionne en ligne de commande, ce qui n'est pas évident à utiliser pour le commun des mortels pas très expérimentés en informatique. C'est donc pourquoi il serait idéal d'avoir une interface visuelle.

Avec une interface, il serait plus facile par exemple de choisir les limites à prendre pour les images. Ou encore d'avoir un avant goût du résultat de sortie pour faire de meilleur choix pour les paramètres.

10.4. Amélioration des algorithmes

Il pourrait être intéressant par exemple de se pencher sur la partie d'analyser des voxels et peut-être de produire plus rapidement la sortie à l'aide des cubes déjà calculer.

De plus, il y a aussi la technique de « *mapping* » utilisée pour la gestion des coordonnées 3D qui pourrait être améliorée. Probablement sur l'allocation mémoire qui est demandée. Même s'il y a déjà un recyclage pour éviter de nouvelles allocations inutiles, il pourrait y avoir des astuces pour provoquer plus rapidement ce recyclage.

Finalement, une technique qui permettrait plus facilement ou plus rapidement de reconnaître un voxel ferait gagner énormément de temps lors de l'exécution. Mais il est difficile à considérer que c'est possible à trouver puisqu'il faut au minimum comparer les huit sommets avec les limites.

11. Conclusion

Le projet peut être considéré comme étant terminé. Les objectifs de base ont été accomplis. Et puisque les objectifs principaux ont été atteints assez rapidement, il y a eu place pour ajouter

des éléments qui rendaient plus intéressant les résultats à présenter. Mais encore là, comme mentionner dans la section « *Développements futurs* », il y a plusieurs amélioration possible.

Je crois qu'il faut voir ce projet comme le départ à un projet beaucoup plus important. Ou encore plusieurs projets avec celui-ci comme source.

Il est important aussi de considérer qu'avec la présence de plus en plus d'outil en milieu médical, le format DICOM devra être de plus en plus connu. Ce qui va permettre plus facilement l'utilisation des données que le format contient ou la création de nouveau outil pour mieux comprendre les résultats obtenus.

Puisque ce projet contient une analyse du format, ainsi qu'une interaction avec celui-ci, il pourrait permettre à d'autres étudiants de se familier avec cet aspect du monde médical. Ces étudiants pourront avoir une expertise qui n'est pas encore enseignée pour le moment dans les universités. C'est donc ces connaissances qui vont contribuer à l'avancement de nouvelle technologie médicale.

12. Annexe

12.1. La syntaxe des algorithmes

Pour décrire les algorithmes, le style de pseudo code choisi est inspiré du langage Python et de certaines syntaxes mathématiques. Pour les segments de code, tel que des fonctions, des boucles ou des conditions, le corps correspondant à ce segment est tout ce qui est indenté vers la droite de deux espaces.

Voici donc la description des mots clés et de certain opérateur :

- **def** :
  - la définition d'une fonction.

- $\Sigma \leftarrow T$:
  - assigne la valeur de T dans Σ.

- **foreach** σ **in** Σ [**where** K] :
  - une boucle qui parcourt chaque élément d'un tableau. Le tableau est Σ et les éléments sont placés, chacun leur tour, dans σ. La dernière partie, est une condition pour accepter l'élément σ avant de parcourir la boucle, si σ échoue la condition K, il est ignoré et c'est le tour du prochain élément. Si la condition n'est pas marquée, puisqu'elle est optionnelle, chaque élément passe la condition avec succès.

- **if** K :
  - test la condition K, si celle-ci réussie, le corps est exécuté, dans le cas contraire, il est ignoré.

- **range**(A, N) :
  - créer un ensemble contenant l'ensemble de valeur entre A et N, inclusivement. Donc, range(A, N) peut être exprimer mathématiquement par [A; N]

- $\Sigma \vee T$:
  - l'union de l'ensemble Σ avec l'élément T. L'élément T est ajouté à la fin de Σ.

- Σ_i :
  - prendre l'élément i dans l'ensemble Σ.

- $\{\}$:
  - représente l'ensemble vide.

12.2. Outils de développement

Le compilateur :

Le principal outil utilisé durant ce projet a été naturellement le compilateur. Le compilateur « *gcc* » a été choisi pour sa gratuité et son aspect multi plateforme. À cause de cet aspect, ce sont les mêmes instructions pour compiler sur *Windows* ou sur une distribution *Linux* ou *Unix*. Afin que les atouts de « *gcc* » soient utilisés au maximum, les librairies ont été nommées en respectant les standards de celui-ci.

L'éditeur VRML :

Puisque avant ce projet, je n'avais pas touché au VRML, j'ai utilisé le logiciel VrmlPad pour faire les premiers essaient avec des modèles 3D très simples. Par la suite, ce logiciel m'a été utile pour aider à corriger les premiers fichiers VRML produits.

12.3. Compilation

Pour aider à créer les librairies mentionnées plus haut, les deux parties du projet possèdent leur propre « makefile ». Alors, il suffit d'aller en ligne de commande dans le répertoire de la partie du projet voulu et d'exécuter « make ».

Chaque « *makefile* » possède deux options de compilation :

make all :

Ceci génère les librairies par défaut. En fait, l'option « *all* » est prise par défaut, il n'est donc pas nécessaire de l'écrire.

make debug :

Ceci génère la version « *debug* » des librairies. Cette version possède l'affichage de la trace pour facilité le déboguage autant en période de développement que pour un futur utilisateur de la librairie dans un de ses logiciels.

Tous les messages d'erreur sont affichés par la sortie standard pour les erreurs (*en C, c'est «* stderr *» et en C++ c'est «* cerr *»*). Le nom des librairies est légèrement différent, il y a « *_debug* » avant le point et le type d'extension du fichier.

Pour produire un exécutable, il suffit d'écrire la ligne suivante (*les parties entre* [] *sont des options à choisir pour la compilation et celle suivit d'une étoile signale qu'elles doivent être écrites*) :

gcc [-o *exécutable*] [-DLANG=*langage*] [-D_DEBUG] main.c language.c [*librairie*]

[-o *exécutable*] :

Cette option permet de choisir le nom de l'exécutable. Si celle-ci n'est pas écrite, le nom de l'exécutable sera « *a.exe* » pour *Windows* et « *a.out* » pour une plateforme *Linux* ou *Unix*.

Pour la première partie du projet, le nom de l'exécutable suggéré est « *dicom2raw* » et pour la seconde, « *raw2vrml* ».

N.B. : l'extension « *.exe* » doit être ajoutée à la fin du nom de l'exécutable pour *Windows*.

[-DLANG=*langage*]:

Ici, l'utilisateur a le choix de compiler le logiciel selon la langue de son choix. Présentement, il n'y a que le français et l'anglais d'implémenter, mais comme mentionné plus haut, il est très facile d'y ajouter une langue de plus. Pour compiler en français, il suffit de remplace *langage* par la valeur **FR** et **EN** pour l'anglais. Par défaut, si ce n'est pas mentionné, le logiciel sera compiler en anglais.

[-D_DEBUG] :

Cette option, si elle est écrite, doit être utilisée avec une librairie de déboguage. Pour faciliter la gestion des messages d'erreur, la sortie standard d'erreur est redirigée vers un fichier texte. De cette façon, il est possible de visionner la trace du déboguage plus facilement.

[*librairie*]* :

Il est absolument nécessaire de mentionner cette option puisque c'est la librairie utilisée. Le nom des librairies est mentionné plus haut. Et si l'option précédente est activée, il faut utiliser la librairie en déboguage.

Finalement, voici deux exemples pour la compilation :

1. Compilation de la première partie, en langue française sur *Windows* :
    - **gcc -o** dicom2raw.exe **-D**LANG=FR main.c language.c libdicom2raw.a
2. Compilation de la deuxième partie, en mode déboguage pour *Linux*:
    - **gcc -o** raw2vrml **-D**_DEBUG main.c language.c libraw2vrml_debug.a

N.B. : Il est important de noter que chaque partie possède ses propres fichiers « *main.c* » et « *language.c* ». Donc, les exemples de compilation précédents sont exécutés dans le répertoire de leur partie.

12.4. Guide d'utilisateur

Comme mentionner dans la partie précédente, les deux parties possèdent une interface, en ligne de commande pour utiliser les librairies.

12.4.1. dicom2raw

dicom2raw <*input file*> <*options*>

-o <*fileOut*>
Envoie la sortie dans le fichier <*fileOut*>

-stdout
Envoie la sortie dans la sortie standard

-L <*files*>
Prend la liste d'entrée <*files*> où chaque nom de fichier est séparé par un point-virgule

-v
Afficher la version de dicom2raw

--help
Afficher cette information

-c <*level*>
Niveau de compression, <*level*> doit être un entier entre 0 et 4

Si l'utilisateur veut par exemple convertir quatre fichiers DICOM (*dicom1, dicom2, dicom3, dicom4 dans le répertoire* ***images***) contenant chacun une image en un fichier de données brutes, nommé « *mondicom.raw* », avec une compression de niveau 1, il doit écrire la ligne suivante :

dicom2raw -o mondicom.raw **–L** images/dicom1;images/dicom2;images/dicom3;images/dicom4 **-c** 1

12.4.2. raw2vrml

raw2vrml <*input file*> <*options*>

-o <*fileOut*>
Envoie la sortie dans le fichier <*fileOut*>

-stdin
Prend l'entrée de donnée dans l'entrée standard

-v
Afficher la version de raw2vrml

--help
Afficher cette information

-s <*scale*>
Le niveau d'étirement sur l'axe des Z

-a
Accepte à l'intérieur des limites

-min <*mini*>

La valeur minimum la limite de couleur*

-max <*maxi*>
La valeur maximum la limite de couleur*

***obligatoire**

Si l'utilisateur veut convertir son fichier de données brutes « *mondicom.raw* » en un fichier VRML en considérant seulement les pixels possédants un niveau de gris (*entre 0 et 255 où 0 est noir et 255, blanc*) entre 50 et 100.

raw2vrml -o mondicom.wrl mondicom.raw **-min** 50 **-max** 100 **-a**

12.5. Bibliographie

1. William E. Lorensen & Harvey E. Cline, "*Marching cubes: A high resolution 3D surface construction algorithm*", SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques, 1987, page 163-169, ACM Press

2. Rikk Carey, Gavin Bell & Chris Marrin, "*ISO/IEC 14772-1:1997, Virtual Reality Modeling Language, (VRML97)*", 1997

3. NEMA PS3, "*Digital Imaging and Communications in Medecine (DICOM), Part 5: Data Structure and Encoding*", 2004

4. NEMA PS 3.6-2004, "*Digital Imaging and Communications in Medecine (DICOM), Part 6: Data Dictionary*", 2004