Caractérisation et quantification d'anévrismes sur des modèles volumiques

Sahar Hassan

Caractérisation et Quantification d'Anévrismes sur des Modèles Volumiques

Sahar HASSAN
Superviseurs : Franck Hetroy et François Faure

Master Recherche Image Vision Robotique
Promotion 2006/2007, Mars - Juin 2006
EVASION/LJK INRIA

TABLE DES MATIÈRES

Remerciements

Merci à mes encadrants pour leur experience, leur patience et le temps, qu'ils m'ont accordé sans parcimonie.

Merci en particulier à François Faure de m'avoir aidé à acquérir les connaissances nouvelles nécessaires dans mon travail.

Merci en particulier à Franck Hetroy de m'avoir aidé à rédiger ce rapport et d'avoir corrigé ma langue approximative.

Merci à Dr.Olivier Palombi pour toutes les informations médicales qui étaient nécessaires pour accomplir ce projet.

Enfin, merci à tous les membres de l'équipe EVASION pour leur accueil chaleureux, et la bonne ambiance qu'ils entretiennent.

CHAPITRE 1

Contexte

1.1 Position du Problème

Dans le cadre des maladies des vaisseaux sanguins, les médecins ont besoin de détecter les déformations : dépôts calcaires, étranglements, anévrismes. L'anévrisme est une dilatation localisée de la paroi d'une artère aboutissant à la formation d'une poche de taille variable, communiquant avec l'artère au moyen d'une zone rétrécie que l'on nomme **le collet**. Sa forme habituelle est celle d'un sac, son diamètre pouvant atteindre plusieurs centimètres.

Les dimensions d'un anévrisme, sa forme et son collet jouent un rôle critique dans le choix de traitement : Soit l'embolisation de l'anévrisme à l'aide de coils (petits ressorts en platine), soit l'intervention chirurgicale (trépanation et mise en place d'un clip au niveau du collet de l'anévrisme).

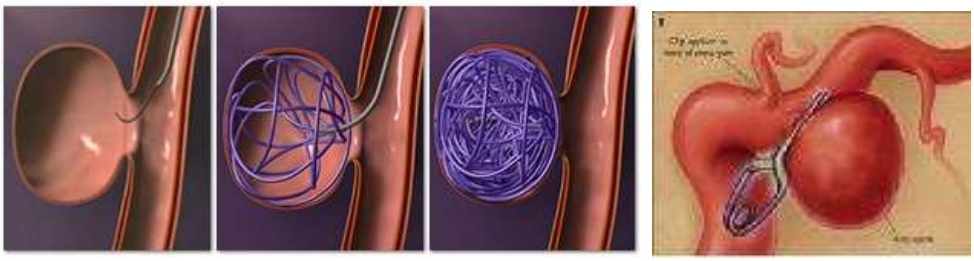


FIG. 1.1 – Traitements possibles : à gauche : embolisation, à droite : clipping `http://www.brainaneurysm.com/aneurysm-treatment.html`

Nos données sont issues d'angiographies par résonance magnétique (ARM), chaque image 2D en niveaux de gris représente une coupe longitudinale du cerveau. Cette technique est spécifique pour visualiser les artères surtout, celles du cou, cerveau et l'aorte abdominale et thoracique. De nombreuses techniques peuvent être utilisées pour engendrer les images, telle que l'administration d'un agent paramagnétique, ou la technique dite de "*flow-related enhancement*", où le signal représente le sang qui vient d'entrer dans le plan de coupe. Grâce à ces techniques, les vaisseaux sanguins sont, généralement, blancs, donc faciles à segmenter.

Mon but est de visualiser l'arbre vasculaire et de détecter les anévrismes, et aussi de définir leurs collets et leurs tailles en minimisant l'intervention d'un médecin. Il y a donc plusieurs étapes à faire :

- **Segmentation** :

  Pour supprimer le bruit et toutes les informations autres que l'arbre vasculaire.
- **Construction d'une représentation volumique** :

  On transforme les données 2D en 3D (d'un pixel à un voxel) pour la visualisation et le calcul.
- **Détection automatique des anévrismes et de leurs collets**
- **Calcul des tailles et formes des anévrismes et de leurs collets** :

  Qui est l'étape la plus importante dans notre travail.
- **Visualisation**

1.2 État de l'Art

L'imagerie médicale est un champ d'étude très important et en plein développement. Ainsi, il existe beaucoup de littérature dans ce domaine.

1.2.1 Segmentation

Segmenter l'arbre vasculaire est une tâche délicate et compliquée. Les techniques de base de segmentation sont classées dans [10] dans quatre catégories :

- Les méthodes orientées par le bord (*edge-oriented*).
- Les méthodes orientées par la région (*region-oriented*) : seuillage, croissance de région, watershed.
- Les méthodes de contours actifs basés sur le modèle : *snakes*, lignes de niveaux.
- Les méthodes hybrides.

Plusieurs algorithmes on été proposés pour la tâche spécifique de segmenter l'arbre vasculaire. L'algorithme proposé dans [5] est semi-automatique, car l'utilisateur doit initialiser un modèle déformable, puis ce modèle est déformé jusqu'à ce qu'il correspond le mieux possible au bord de l'objet. Les résultats dépendent du modèle, et sont pas très satisfaisants au niveaux des bifurcations.

Les auteurs de [1] ont choisi un algorithme de ligne de niveaux. Les lignes sont initialisées par des ballons dans les plus grands vaisseaux, puis les lignes de niveaux se développent. Le nombre de points initiaux n'est pas facile à déterminer.

Dans [13], les auteurs proposent un algorithme local de croissance de région, qui commence par segmenter un petit cube pour détecter un segment du vaisseau, puis la détection du cube(s) local suivant est basée sur le résultat de la segmentation. Ce processus est répété jusqu'à completer la segmentation. Cet algorithme est robuste même quand l'intensité change, mais il peut pas détecter toutes les branches.

Flasque et *al.* [2] ont proposé une méthode pour segmenter l'arbre vasculaire du cerveau, cette méthode détecte d'une manière itérative les lignes de centre de voxels candidats. On décide la candidature d'un voxel par combinaison de correction d'intensité, filtrage de diffusion pour supprimer le bruit, et de croissance de région

pour isoler les voxels. Cet algorithme fournit des informations topologiques sur les vaisseaux mais pas d'informations morphologiques (les diamètres des vaisseaux) ; le seuil statique (le même partout) utilisé est également une limite de cette méthode.

Trouver les meilleurs paramètres pour segmenter l'image peut se faire en utilisant un atlas comme dans [11], où l'atlas est construit à partir des images ARM de contraste de phase (*PC MRA*) en trois étapes : extraction de connaissance, enregistrement et fusion de données. Puis en utilisant les informations anatomiques, et l'atlas comme modèle, on est capable de trouver les paramètres pour le processus de segmentation. De toutes façons, l'utilisation d'un atlas est sensible aux différences inter-individus et intra-individus.

Dans ce projet, la segmentation n'est pas la tâche principale. J'ai choisi donc, un simple seuillage avec la possibilité de régler le seuil par l'utilisateur. L'utilisation d'un atlas est envisagée à l'avenir pour améliorer la segmentation.

1.2.2 Extraction de la ligne de centre

Pour **détecter les anévrismes**, nous avons choisi d'extraire le squelette de l'arbre vasculaire. Dans un premier temps, on a extrait l'axe médian, en se servant de l'algorithme proposé dans [6]. L'idée est de trouver les points dont la distance au bord est maximale.

Mais, l'axe médian n'est pas exactement ce que l'on attend, parce que dans certaines régions, cet axe est une surface (voir section 2.1). On est passé donc vers l'extraction d'une "ligne de centre".

On peut classer les méthodes existantes pour l'extraction de la ligne de centre dans les catégories suivantes :

1. **Les méthodes manuelles** :

   Dans ces méthodes, l'utilisateur choisit manuellement les points clés, et l'algorithme fait l'interpolation entre ces points. Les résultats sont très subjectifs au choix des points, qui est une tâche fastidieuse et non précise.

2. **Amincissement topologique** :

L'idée principale derrière ces méthodes est l'épluchage successif du volume jusqu'à l'arrivée à la singularité. Il s'agit d'enlever les points dits simples, qui sont les points dont la supression ne change pas la topologie. Ces points peuvent être supprimés en parallèle [8] ou en séquentiel [3]. Les résultats sont normalement des lignes de centre : homotopiques (par construction) avec l'objet, fines, et bien représentatives de l'objet (si les points simples sont bien caractérisés) mais pas forcément au centre.

3. **Gradient *zero-crossing*** :

   Dans [7], les auteurs se servent du gradient et des valeurs propres de la matrice Hessienne pour interpoler les *zero-crossing* du vecteur de gradient de valeurs de gris dans la direction de la section du vaisseau. Cette idée est valable surtout en couleur continue (directement sur les images, où les différentes valeurs de gris permettent de trouver la direction du vaisseau) et non pas discrète (sur les images segmentées qui sont binaires).

4. **Carte de distance** :

   Les algorithmes de *distance mapping*, consistent à associer à chaque voxel ou pixel une distance. Cette distance peut être une distance à un point source comme dans l'algorithme de Dijkstra, ou une distance de chanfrein. Il s'agit ensuite de trouver la ligne de centre qui est la branche la plus longue dans l'arbre construit pour trouver ces distances. Ces méthodes sont rapides et donnent une ligne connexe et singulière, mais qui coupe les virages. Pour résoudre ce problème, les auteurs dans [12] trient les voxels selon l'inverse de leurs distances au bord, pour obliger la ligne à être plus au centre. Une idée très semblable est employée dans [4].

   L'algorithme que j'ai développé tombe dans cette catégorie, et il est détaillé dans le deuxième chapitre.

Nous avons choisi de travailler directement sur les voxels plutôt que sur une triangulation représentant la surface de l'objet, et calculée, par exemple, grâce à l'algorithme des marching cubes [9], car l'étude est plus simple et nous apporte

suffisamment d'information. De plus, en travaillant sur les données brutes, nous n'introduisons aucune approximation.

1.3 Contributions

Ce projet est concentré sur la détection des anévrismes, leurs collets, et leurs quantifications en minimisant l'intervention de l'utilisateur. En supposant que la segmentation est bien faite, la donnée de départ est donc un ensemble connexe de voxels qui représentent l'arbre vasculaire.

Puis, et à partir de cet ensemble de voxels, je procède à la détection des anévrismes. Pour cela, l'extraction d'une ligne du centre m'est apparu très utile. Car avec cette ligne je peux calculer le diamètre approximatif des vaisseaux, et puis m'en servir pour détecter les anévrismes. Pour faire cette extraction, je me suis inspirée de l'algorithme proposé dans [12], j'ai implementé cet algorithme avec deux modifications : une pour extraire une ligne de centre robuste, et l'autre pour extraire des branches dont les jonctions sont les plus "naturelles" possibles. Ces modifications sont détaillées dans le chapitre suivant.

Ensuite, je me sers de la ligne de centre, les branches, leurs diamètres pour détecter les anévrismes d'une façon totalement automatique. Car, généralement l'anévrisme sera détecté comme étant une branche de l'arbre vasculaire, mais qui est différente d'une branche normale. Je vais expliquer cette différence et comment s'en servir pour détecter les anévrismes dans le troisième chapitre.

Une contribution importante est la détection du collet de chaque anévrisme trouvé automatiquement. Je me suis servi de l'anévrisme trouvé pour détecter son collet. Les détails sont expliqués au quatrième chapitre.

Enfin, pour la visualisation, j'ai choisi *Open Inventor*, en permettant à l'utilisateur de choisir une zone d'intérêt et de voir (ou pas) un certain groupe de voxels (les voxels de : la surface, la ligne de centre principale, les branches, les anévrismes, les collets), voir annexe B.

CHAPITRE 2

Extraction du Squelette de l'Arbre Vasculaire

2.1 L'Axe Médian

Dans un premier temps, on a pensé à extraire l'axe médian, car cet axe peut être une représentation compacte, connexe de la topologie de l'arbre vasculaire, et peut nous aider à détecter les anévrismes. Pour ce faire on a appliqué l'algorithme de *Fast Marching Method* présenté dans [6].

Avant de décrire l'algorithme, je voudrais préciser les points suivants :

- On travaille sur les points qui sont les sommets des voxels, et pas sur les voxels eux-mêmes.
- Quand on parle de voisins, on parle des six voisins perpendiculaires.
- On considère un sommet comme un sommet de la surface de l'objet si un de ces six voisins n'existe pas.
- Quand on parle de la distance entre deux points, c'est toujours la distance Euclidienne.
- Pour chaque sommet on stocke la distance à la surface, et sa "caractéristique la plus proche" (*closest feature*) qui est le sommet de la surface le plus proche à ce sommet.

On peut décrire cet algorithme comme suit :

D'abord, on marque comme *alive* tous les sommets de la surface, et on les insère

dans une list triée selon la distance, cette liste est initialement vide. On met leur distance à zéro et leur *closest feature* à eux même.

Puis, on marque comme *close* tous les sommets qui sont voisins des sommets *alive*. Tous les sommets restants sont marqués comme *far*.

Après l'initialisation précédente on commence la boucle suivante :

1. Parmi les sommets *alive* on prend le sommet dont la distance est minimum, on va l'appeler C.
2. On teste chaque voisin B de C sauf les voisins marqués comme *alive*, et si la condition :

   distance[B] + distance(C,B) $<$ distance[B]

   est satisfaite, on modifie le voisin B par le marquer comme *close*, changer sa distance et l'insérer dans la liste triée.
3. Si le sommet C n'arrive pas à modifier un seul voisin, on dit qu'il appartient à l'axe médian.
4. On marque le sommet C comme *alive*.
5. On répète les étapes précédentes jusqu'à que la liste soit vide.

La figure 2.1 présente le résultat de cet algorithme.

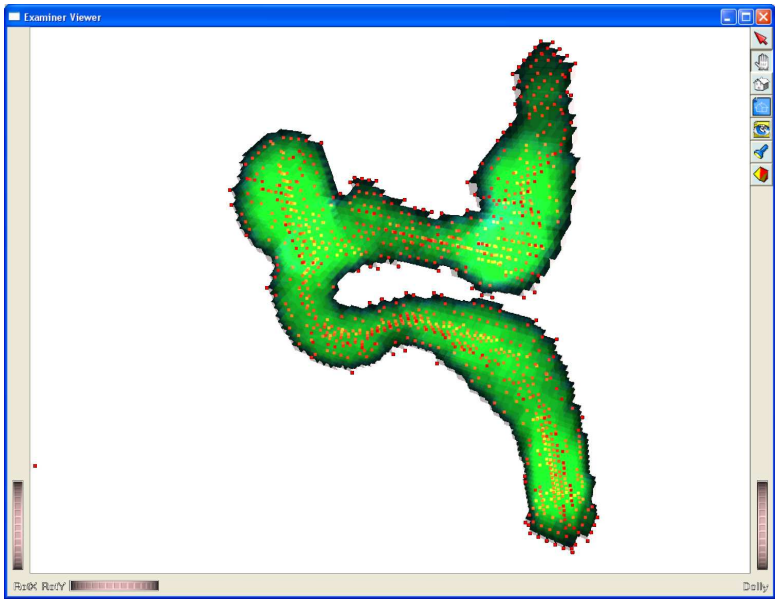


FIG. 2.1 – L'axe Median

Comme on le voit sur la figure 2.1 le résultat n'est pas une représentation compacte et connexe de l'objet. En effet, l'axe médian est un groupe de points, qui ne constitue pas une ligne connexe sur laquelle on peut se baser pour calculer approximativement le diamètre du vaisseau sanguin. Au contraire, ces points forment des surfaces qui ne peuvent pas être très utilisables pour nous.

Pour bien voir les surfaces dont je parle, j'ai exécuté l'algorithme sur un groupe d'images où les pixels "noirs" forment un parallélépipède comme illustré sur la figure 2.2.

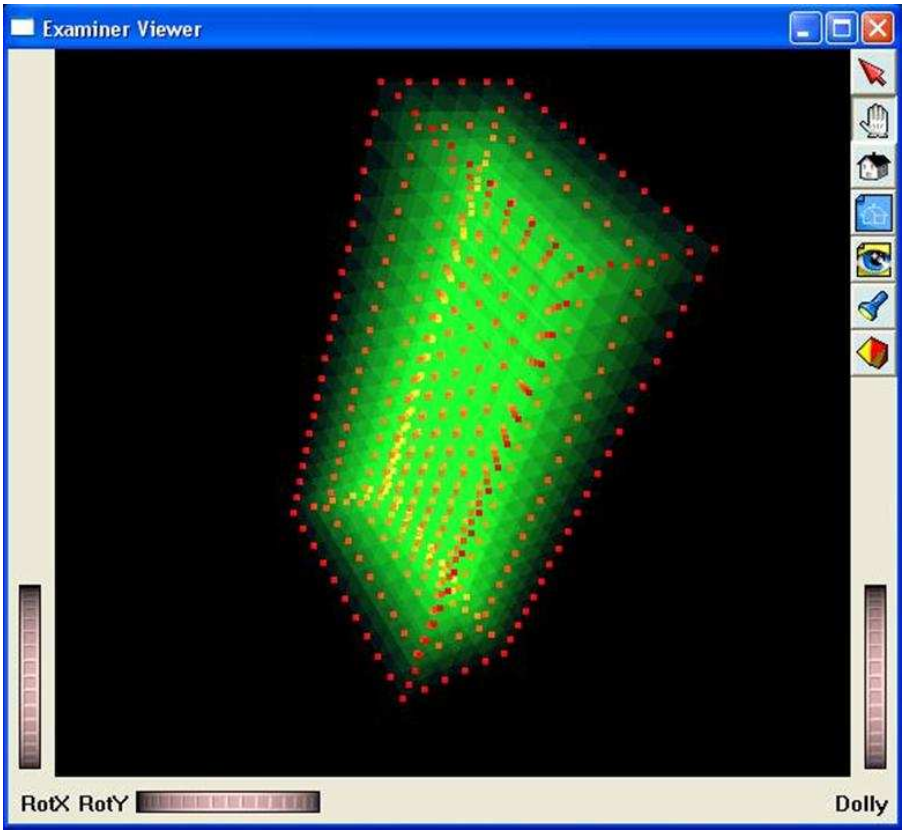


FIG. 2.2 – L'axe médian d'un parallélépipède

On voit sur la figure 2.2 que les points de l'axe médian forment des circonférences des parallélépipèdes qui se forment en enlevant chaque fois la surface du parallélépipède, quelque chose comme un épluchage. À la fin, pour le dernier parallélépipède, les points forment un rectangle (une surface).

Ces résultats peuvent être expliqués par le fait que la définition de l'axe médian est : l'ensemble de centres (points) des sphères qui sont tangentes à l'objet en deux points ou plus. Nous cherchons des voxels connexes qui gardent la topologie de l'objet. Nous avons donc pensé à utiliser plutôt la notion de ligne de centre.

2.2 La Ligne de Centre

Dans [12], est expliqué un algorithme pour extraire une ligne de centre. L'algorithme est basé sur le calcul de l'arbre des plus courts chemins d'un voxel source à tous les autres voxels selon l'algorithme de Dijkstra avec une modification. Cette

modification est que, au lieu de trier les voxels selon leur distance à la source, les auteurs proposent de les trier selon l'inverse de leur distance à la surface. On verra ça plus clairement lors de l'explication de l'algorithme.

Le voxel *source* est choisi au début de l'algorithme soit automatiquement soit par l'utilisateur. Pour minimiser l'intervention de l'utilisateur, j'ai choisi de trouver ce voxel automatiquement. Il est calculé en prenant un voxel V au hasard, puis en prenant comme source le voxel le plus loin de V. De cette façon, on est sûr que la source est une extrémité de l'arbre. Ensuite on construit l'arbre vasculaire.

Le schéma 2.3 explique comment construire l'arbre de Dijkstra.

Pour comprendre le schéma notons que :

1. *DFS* correspond à la distance à la source
2. *DFB* correspond à la distance à la surface
3. *DIS(B,C)* correspond à la distance Euclidienne entre les centres de deux voxels B et C

Puis, on peut remarquer que, à la différence de l'algorithme précédent, dans cet algorithme :

- On travaille sur les voxels et non les sommets
- On regarde les 26 voisins et non les 6

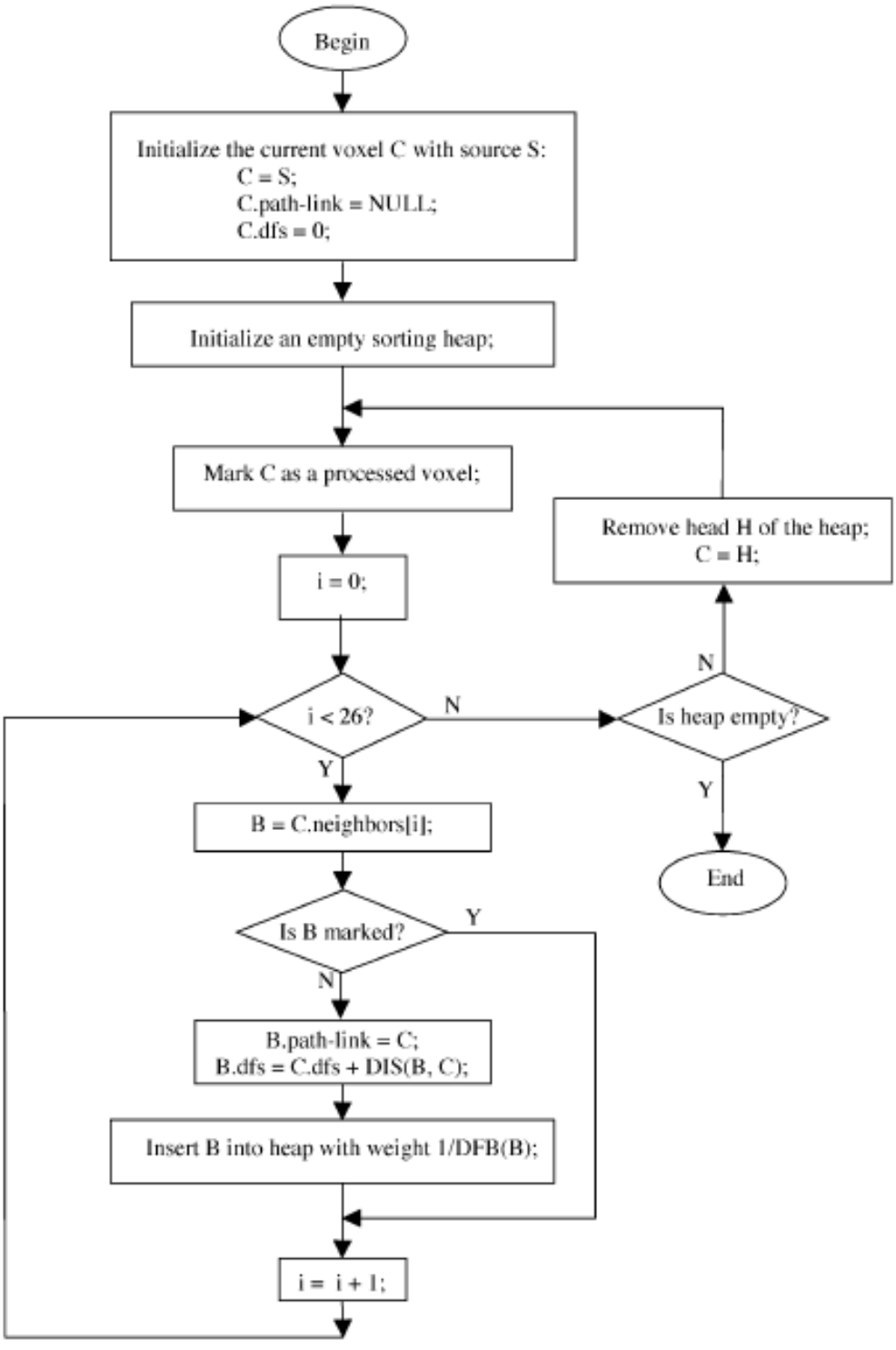


FIG. 2.3 – Schéma de l'extraction de l'arbre de Dijkstra d'après [12]

Après la construction de l'arbre, on définit le voxel *final* comme le voxel le plus loin de la *source*.

Cependant, l'algorithme tel qu'il est décrit dans [12] est erroné et ne donne pas une ligne de voxels passant au centre des vaisseaux. En effet, il manque un test avant de modifier un des voisins d'un voxel. Ce test est le suivant :

$$DFS[voxel] + DIST(voxel, voisin) < DFS[cevoisin]$$

L'algorithme modifié de la sorte donne un résultat plus satisfaisant, comme l'illustrent les figures 2.4 et 2.5.

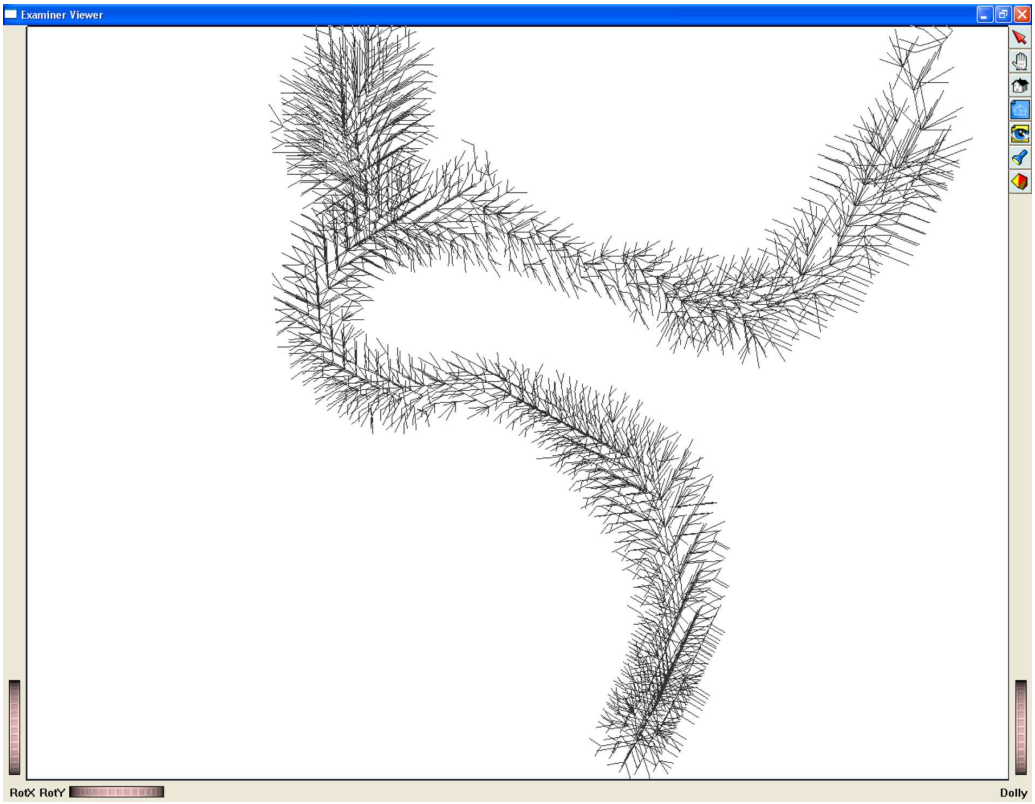


FIG. 2.4 – L'arbre construit après modification de l'algorithme

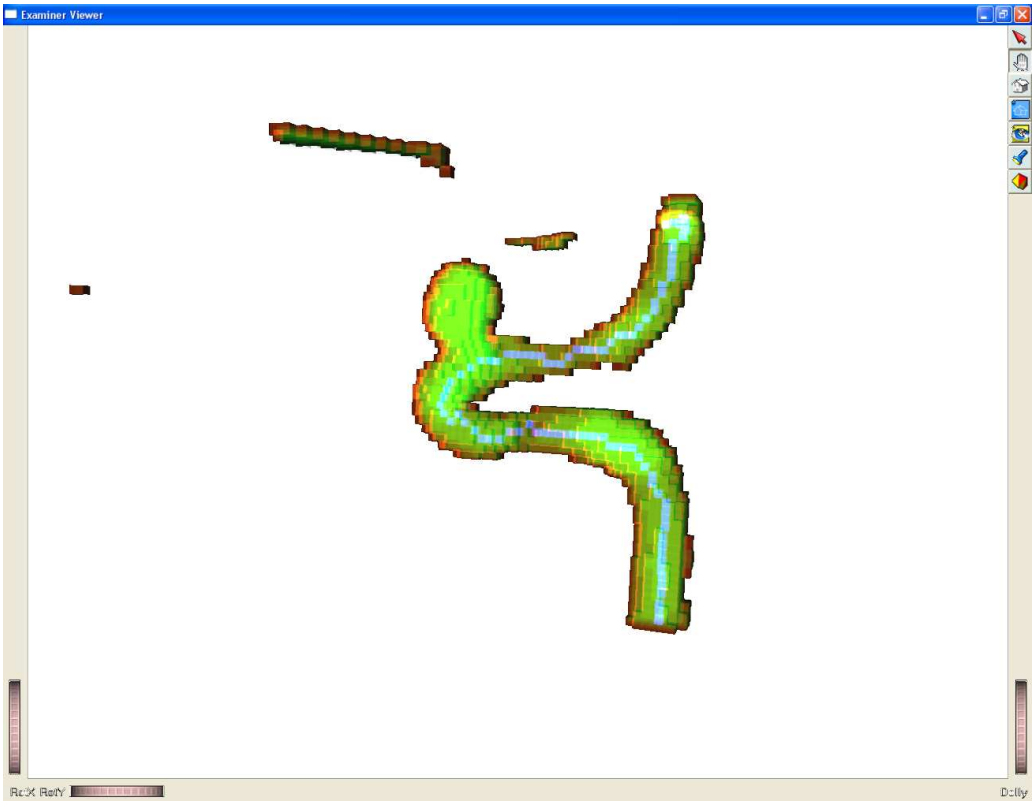


FIG. 2.5 – La ligne de centre (en bleu)

Cependant, les distances calculées selon l'arbre sont très différentes des distances euclidiennes. En effet, il existe des branches de l'arbre qui sont parallèles à la ligne du centre. D'autre part, l'arbre qu'on construit sera aléatoire dans les cas extrêmes ou symmétriques, car plusieurs voxels auront le même profondeur, le voxel choisis à chaque itération sera donc aléatoire. On voit sur la figure 2.6, l'arbre construit pour une seule image 2D.

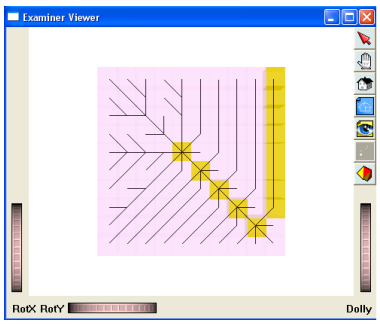


FIG. 2.6 – l'arbre et la ligne de centre pour une seule image

Afin de résoudre ces problèmes, je propose de changer la fonction de distance, au lieu de prendre la distance euclidienne à la source, je propose la fonction :

$$d(F, Ch) = \frac{dist(F, Ch)}{1 + f * (DFB[F] + DFB[Ch])} \tag{2.1}$$

Comme la fonction de distance est proportionnelle à l'inverse des profondeurs des voxels, ceci garantit que les voxels plus profonds seront prioritaires, et cela garantit que la ligne de centre soit au centre (ne coupe pas les virages). En même temps, cette fonction est proportionnelle à la distance euclidienne entre les voxels, qui garantit que entre deux voxels qui ont la même profondeur, on va choisir le plus proche, contrairement à la méthode proposé dans [12], qui va tirer au sort entre les voxels de même profondeur.

Dans le cas particulier d'une seule image, le résultat reste le même, car dans une seule image tous les voxels ont la même profondeur (nulle), donc la fonction n'est pas changée.

Je propose donc également de ne plus trier les voxels selon l'inverse de leur profondeur, mais selon leur distance à la source, comme l'algorithme de Dijkstra original. Comme on le voit sur la figure 2.7, le résultat est plus satisfaisant :

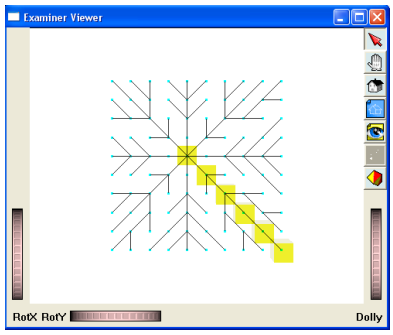


FIG. 2.7 – L'arbre et la ligne de centre pour une seule image en utilisant l'algorithme de Dijkstra

Puis, en utilisant la fonction précédente pour construire l'arbre, on constate que :

1. Pour un f inférieur à 1, la ligne du centre a ,dans quelques endroits, tendance à s'approcher de la surface

2. Pour un f égal ou supérieur à 1, on a toujours la même ligne du centre qui est bien au centre du vaisseau

On voit sur la figure 2.8, la ligne de centre pour différentes valeurs de f.

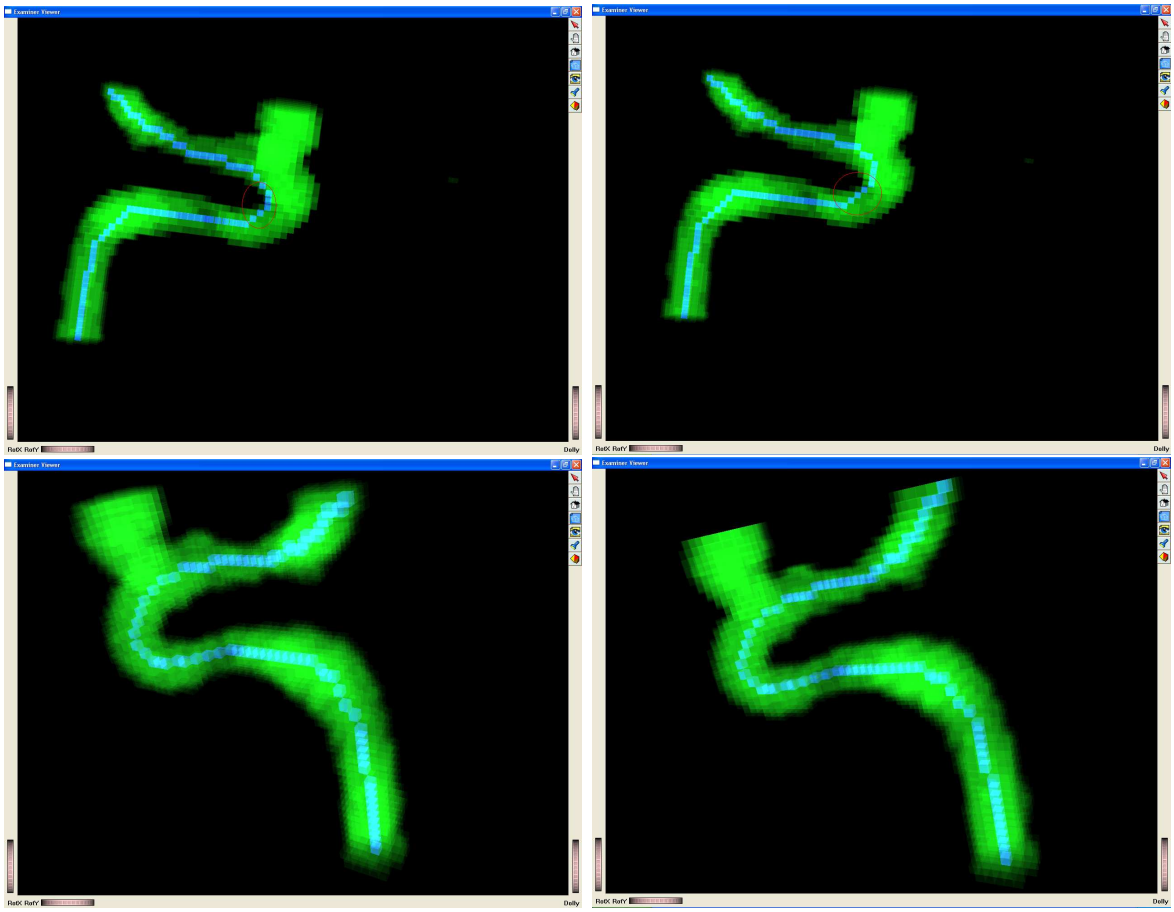


FIG. 2.8 – La ligne de centre pour : *f=0.1* (en haut à gauche), *f=0.2* (en haut à droite), *f=1* (en bas à gauche), *f=10* (en bas à droite)

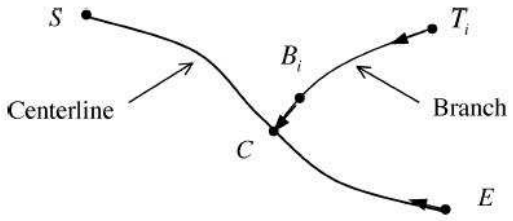
2.3 Extraction des Branches

Après l'extraction de la ligne de centre principale, il faut extraire les branches de l'arbre vasculaire.

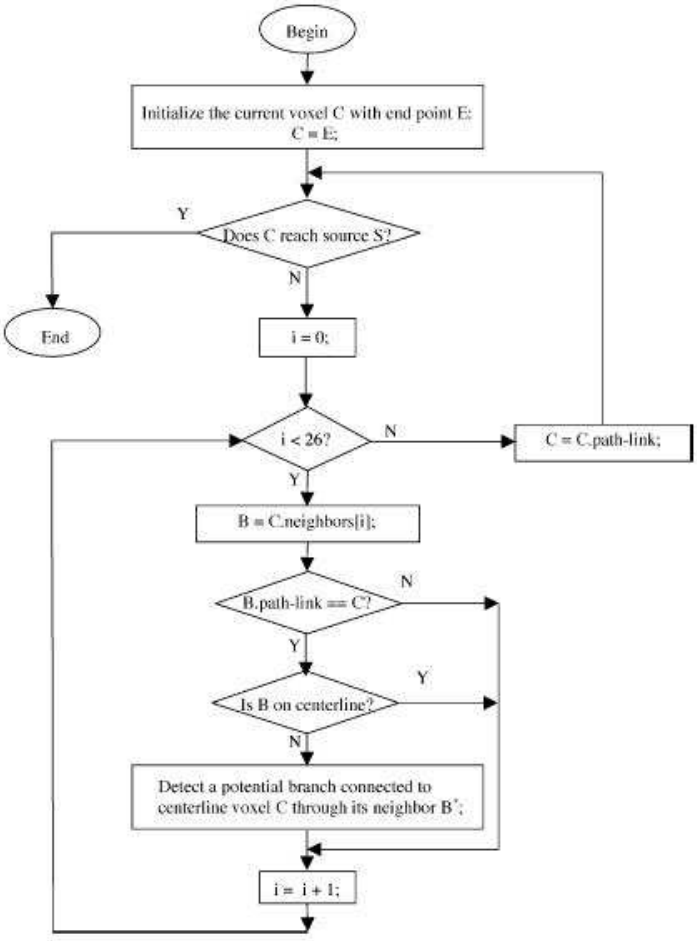
D'abord, j'ai essayé d'appliquer l'idée proposée dans [12], qui semble assez naturelle. L'idée est la suivante :

– Parcourir la ligne de centre à partir de son extrémité (*end point*) vers la source

– Pour chaque voxel Ci, regarder tout ses voisins sauf ceux qui appartiennent à la ligne de centre (24 voisins) et trouver ceux Bi qui sont reliés à ce voxel Ci
– Pour chaque voxel Bi, chercher dans tous les voxels qui sont reliés directement (ses fils) ou indirectement (petit-fils, arrière-petit-fils ...) à Bi le voxel Ti le plus loin à la source, garder ce voxel comme l'extrémité d'une branche
– Si la distance de Ti est plus grande qu'un seuil (longueur minimale de la branche), on trace la branche en descendant dans l'arbre de Ti jusqu'à Ci



Voilà le schéma de cet algorithme tel qu'il est proposé dans [12] :



Le résultat de cet algorithme est visible sur la figure 2.9.

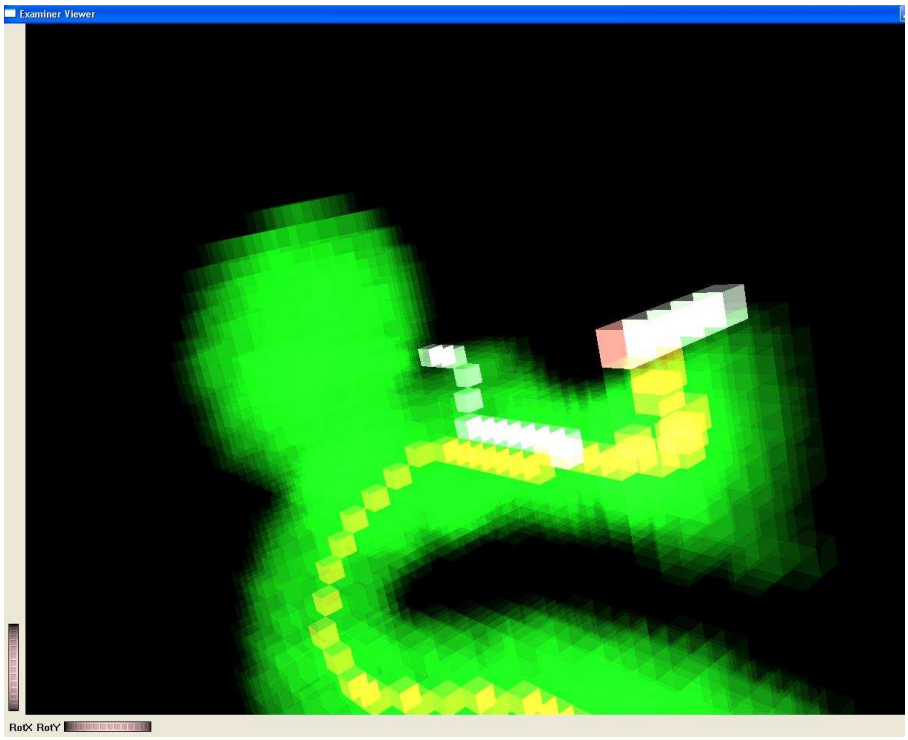


FIG. 2.9 – Les branches selon l'algorithme de [12]

Comme on le voit, le résultat n'est pas bon car : une branche finit parallèlement à la ligne de centre, et les extrémités trouvées ne représentent pas des extrémités de vraies branches de l'arbre vasculaire. On peut expliquer le première problème par le fait que les liaisons entre les voxels dépendent de leurs profondeurs. Le deuxième problème est expliqué par le fait que la distance à la source que l'on utilise pour la construction de l'arbre n'est plus une distance euclidienne, mais elle est affecté par la profondeur des voxels.(voir la fonction de distance 2.1)

2.3.1 Notre Méthode

Pour résoudre le premier problème, c'est à dire obtenir des jonctions naturelles entre les branches de l'arbre vasculaire, on doit s'assurer que les branches de l'arbre de Dijkstra soient le plus perpendiculaires possible à la ligne de centre principale. Je propose donc l'algorithme suivant :

- Après l'extraction de la ligne de centre principale, on met la distance des voxels de cette ligne à la source à zéro.
- On reconstruit un arbre de Dijkstra (en utilisant toujours notre fonction de distance).
- On cherche le voxel le plus loin de la source.
- De ce voxel on descend dans l'arbre jusqu'à rencontrer un voxel de la ligne de centre.
- On répète ces étapes tant que la longueur de branche est assez importante (par exemple : supérieure au diamètre de la ligne de centre principale).

On voit sur la figure 2.10, les résultats de cet algorithme dans deux cas : le premier si on utilise l'arbre original, c'est à dire le premier arbre qu'on a construit, le deuxième si on utilise à chaque fois le dernier arbre construit (pour chaque branche on reconstruit l'arbre). On voit que la bifurcation est beaucoup mieux dans le dernier, car la branche rejoint la ligne principale d'une façon naturelle et non pas parallèle.

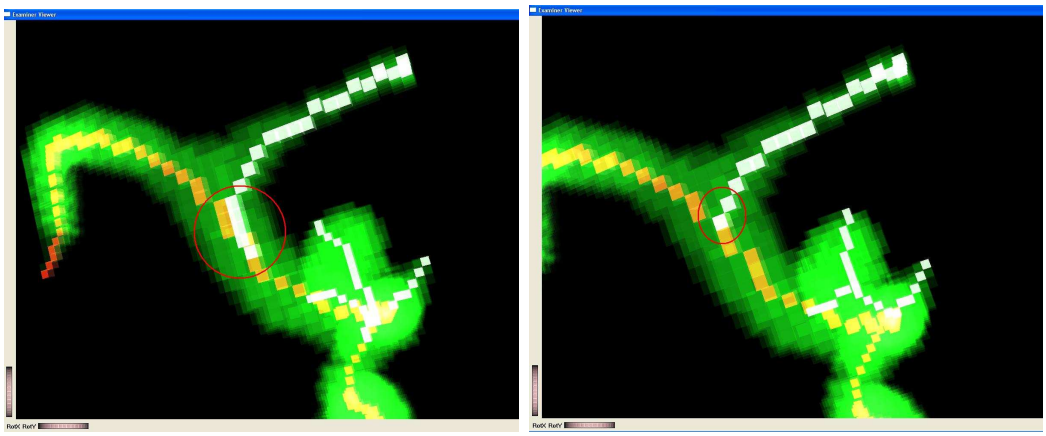


FIG. 2.10 – Les branches en utilisant l'arbre original (à gauche), et le nouvel arbre (à droite)

Pour résoudre le deuxième problème, qui est de trouver les bonnes extrémités pour les branches de l'arbre vasculaire, j'ai choisi de garder notre fonction de distance pendant la construction de l'arbre de Dijkstra (pour être toujours au centre), mais d'utiliser la distance euclidienne, pour la recherche de voxels le plus loin de la source. Ceci nous garantit d'obtenir à chaque étape la plus longue branche possible.

Par exemple, sur la figure 2.11 :

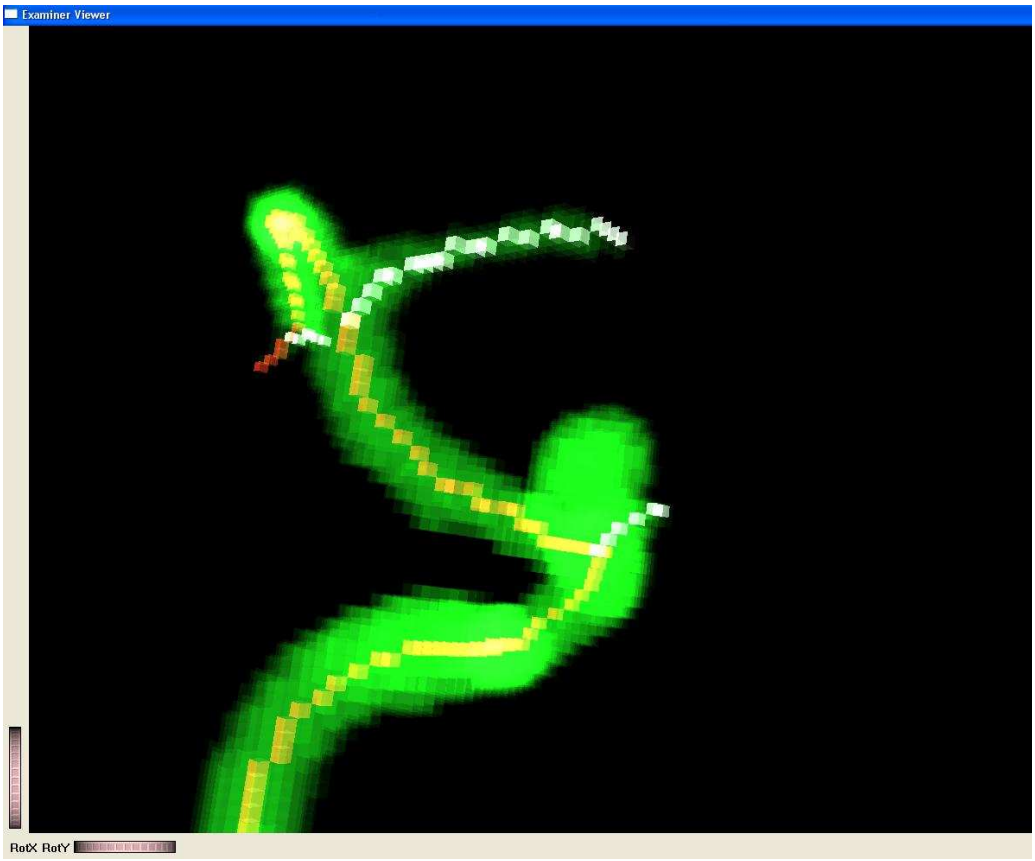


FIG. 2.11 – Les branches si on n'utilise pas la distance euclidienne

On voit qu'il y a deux branches trouvées qui sont évidemment plus courtes qu'une branche qui partait dans l'anévrisme et qui n'est pas encore trouvée. Prendre la distance euclidienne suffit pout trouver les branches dans l'ordre décroissant des longueurs.

CHAPITRE 3

Détection des Anévrismes

La première méthode à laquelle j'ai pensé pour détecter automatiquement les anévrismes repose sur l'idée que la distance des voxels des anévrismes à la ligne de centre est supérieure au diamètre approximatif du vaisseau (la moyenne de distance des voxels de la surface du vaisseau à sa ligne de centre). J'ai donc associé à chaque voxel la branche la plus proche, puis j'ai calculé les diamètres approximatifs des branches. Cette idée donne le résultat visible sur la figure 3.1.

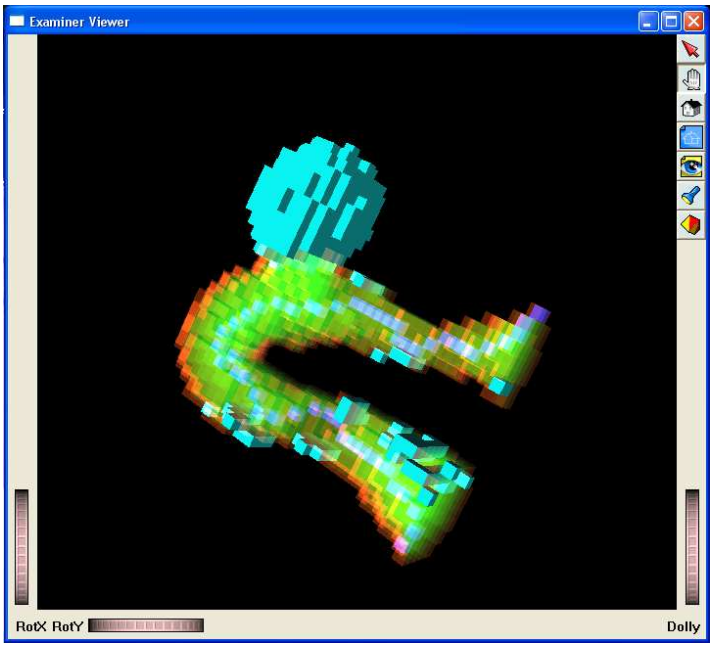
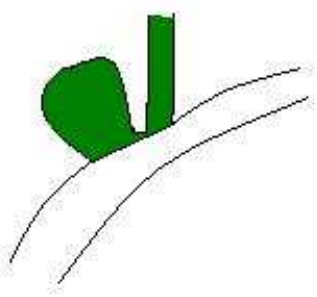


FIG. 3.1 – Les voxels dont la distance à la ligne de centre est supérieure au diamètre approximatif

Comme le montre la figure 3.1, cette méthode de détection possède les inconvénients suivantes :

- Comme le diamètre du vaisseau est calculé approximativement, on voit des voxels du vaisseau qui sont détectés comme dans un anévrisme (*false positive*).
- Si on extrait une ligne de centre principale, on serait obligé de choisir une zone d'intérêt qui ne contient que le vaisseau support de l'anévrisme. Si non, une branche de l'arbre serait considérée comme anévrisme.



- Si on extrait le squelette (la ligne de centre + les branches) de l'arbre vasculaire, il y aura une branche qui part dans l'anévrisme. Donc, utiliser le diamètre approximatif de cette branche ne donnera pas les voxels de l'anévrisme.

Alors, pour que la détection des anévrismes soit automatique et n'exige pas un choix d'une zone d'intérêt, j'ai choisi d'extraire le squelette de l'arbre vasculaire, et puis :

- Trouver les branches qui partent dans les anévrismes.
- Se servir de ces branches pour trouver précisément les voxels qui appartiennent aux anévrismes.

3.1 Trouver les branches des anévrismes

Si on voulait détecter les anévrismes non automatiquement, il suffirait d'extraire les branches successivement, en demandant au médecin de s'arrêter quand une branche correspondant à un anévrisme est extraite. Mais, comme on souhaite une détection entièrement automatique des anévrismes, il faut trouver un critère qui différencie une branche d'un anévrisme d'une branche normale.

En regardant la forme arrondie d'un anévrisme, on peut dire que la distance de la surface de l'anévrisme à sa ligne de centre (la branche qui est dans l'anévrisme)

varie plus que dans une branche normale. À partir de cette idée, on a calculé la déviation de cette distance pour toutes les branches extraites. Si cette déviation est importante (supérieure à un seuil) on peut dire que la branche représente un anévrisme.

Aussi, on regarde si la branche n'est pas connectée à celle d'un anévrisme, car il peut y avoir plusieurs branches dans un seul anévrisme (voir figure 3.2). Dans ce cas, il suffit de trouver la branche principale, et ainsi on est sûr que chaque branche trouvée représente un anévrisme séparé des autres.

Un résultat est visible sur la figure 3.2.

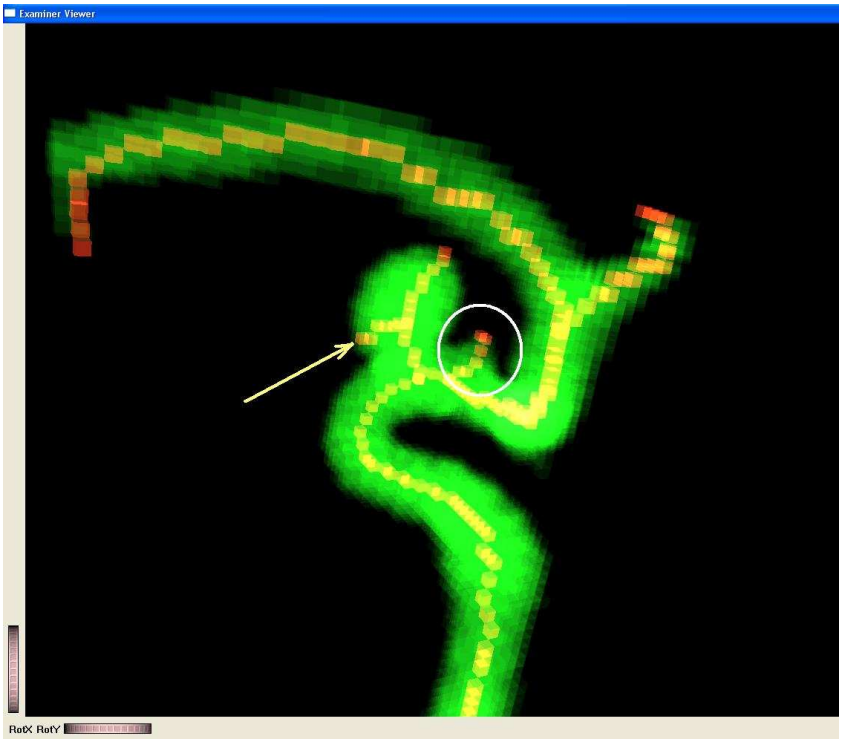


FIG. 3.2 – La détection des branches des anévrismes, la flèche montre une deuxième branche dans le même anévrisme, dans l'ellipse une branche ne représentant pas un anévrisme

Dans cette figure, notre méthode détecte deux branches : une est bien celle d'un anévrisme et l'autre ne l'est pas. C'est compréhensible car cette branche a une forme perturbée (à cause de la segmentation, ou car elle n'est pas assez longue), donc elle a une déviation importante.

Il faut donc trouver un critère plus précis que la déviation. Si on regarde la

distance des voxels de la surface de l'anévrisme à la branche comme une fonction dont l'abscisse est la distance de ces voxels au point de connexion entre cette branche et celle d'avant, on retrouve la forme illustrée dans la figure 3.3.

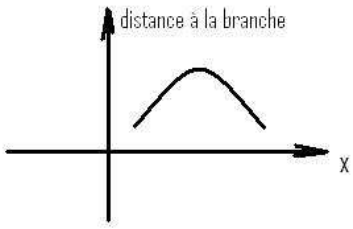


FIG. 3.3 – Fonction de distance

Cette fonction est due à la forme arrondie de l'anévrisme. Elle augmente de l'intérieur vers le milieu de l'anévrisme, puis il diminue vers le bout de l'anévrisme, donc cette fonction a un maximum local. Or, ce n'est pas le cas pour une branche normale, dont la fonction de distance peut varier mais ne peut pas avoir un seul maximum bien distingué.

Je n'ai pas pu appliquer cette idée par manque de temps. Alors, pour bien trouver les branches des anévrismes et continuer/tester les étapes suivantes (détection des anévrismes et de leurs collets), je l'ai fait manuellement, en demandant d'extraire deux branches (dans l'exemple, ceci garantit l'extraction de la branche normale et celle de l'anévrisme).

3.2 Trouver les voxels des anévrismes

Une fois la branche de l'anévrisme détectée, en déduire les voxels de l'anévrisme est relativement simple. Tous les fils des voxels de la branche de l'anévrisme, ainsi que leurs fils, petits-fils ..., appartiennent à l'anévrisme, sauf quelques voxels autour du point de jonction entre la branche de l'anévrisme et son père. On peut voir ces voxels sur la figure 3.4.

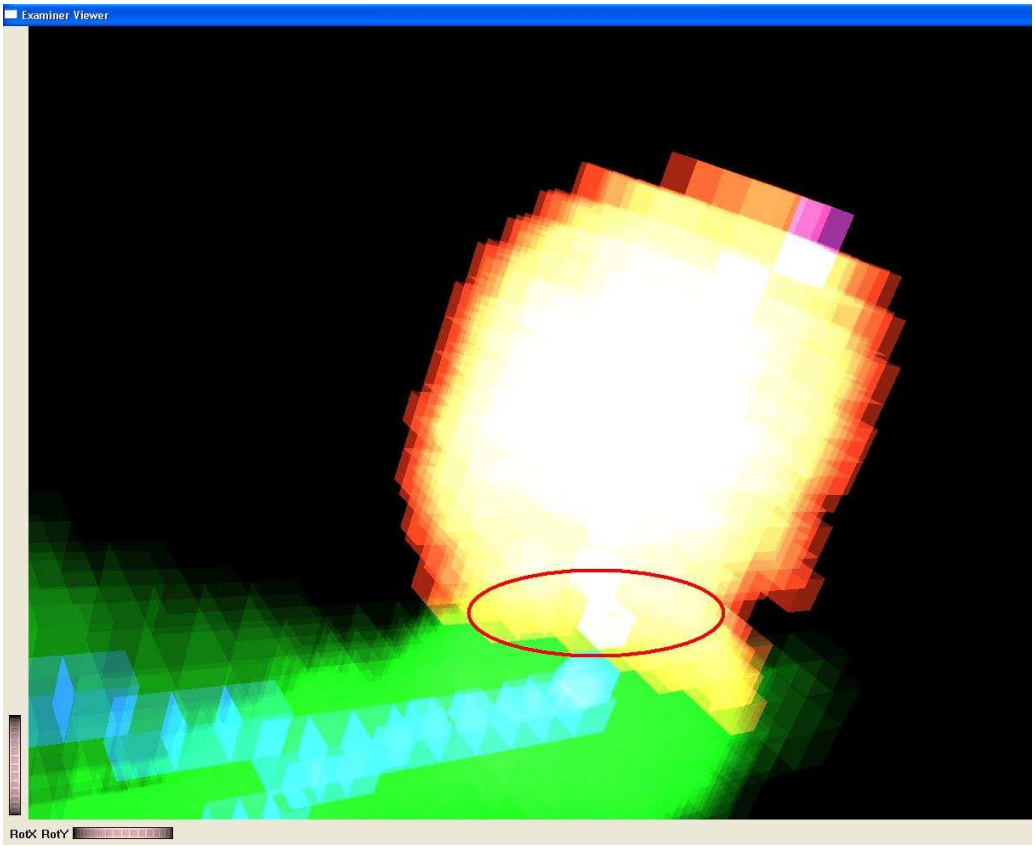


FIG. 3.4 – Les voxels mal détectés

Pour ne pas compter ces voxels, il suffit d'ajouter une condition aux voxels de la branche avant de les considérer comme voxels de l'anévrisme. Cette condition est que la distance de voxel à la branche *père* de la branche de l'anévrisme, soit supérieure à la profondeur du voxel de jonction entre la branche de l'anévrisme et son père.

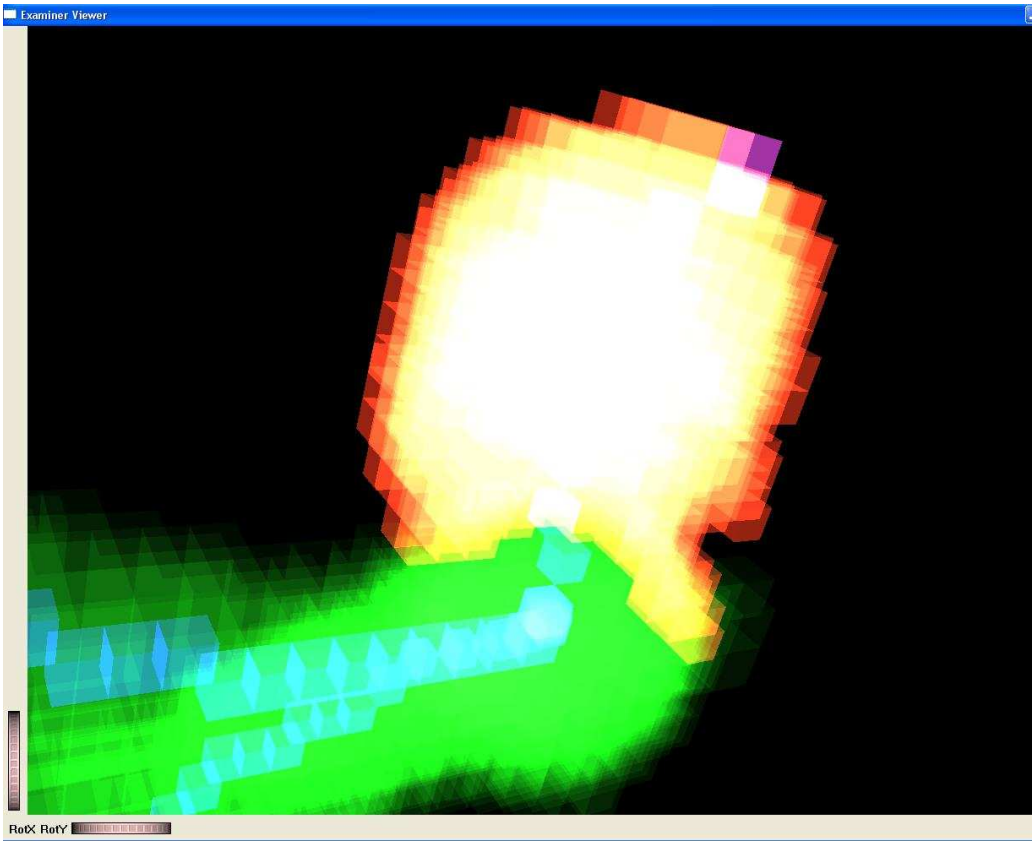


FIG. 3.5 – Les voxels de l'anévrisme avec la condition

Comme on peut le voir sur la figure 3.5, les voxels autour de la jonction ne sont plus considérés comme appartenant à l'anévrisme.

Je vais expliquer dans le chapitre suivant comment utiliser cette approximation de l'anévrisme pour trouver le collet, puis comment se servir du collet pour trouver l'anévrisme plus précisément.

CHAPITRE 4

Détection des Collets des Anévrismes

4.1 Première estimation : paroi

Étant donné l'anévrisme, on peut définir les voxels du collet comme étant les voxels qui ont au moins un voisin qui n'appartient pas à l'anévrisme. En appliquant cette définition on obtient le collet dans la figure 4.1.

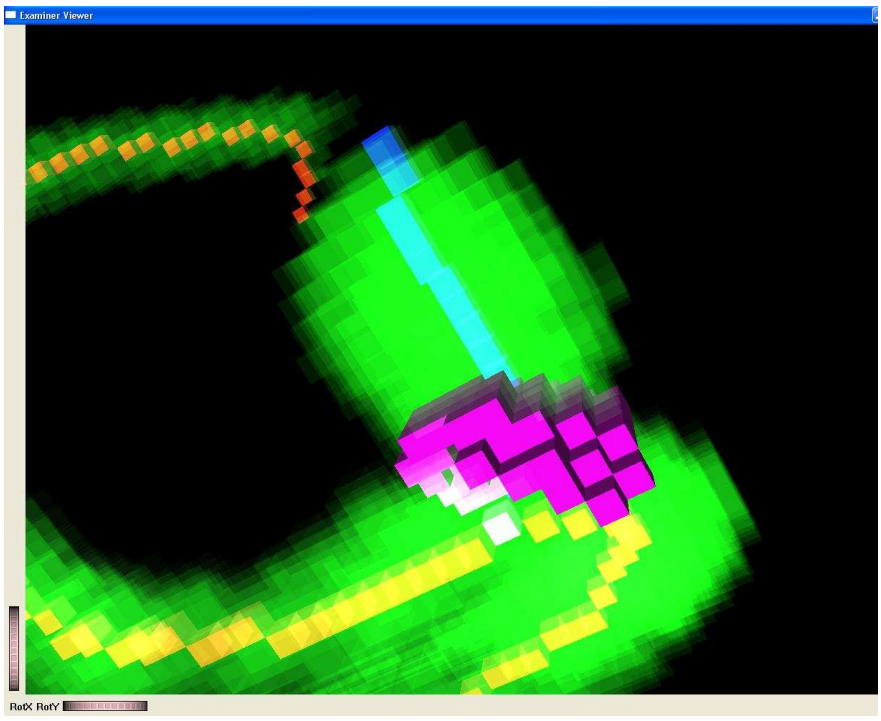


FIG. 4.1 – Première estimation du collet

On constate sur cette figure que la définition appliquée ne correspond pas à la définition médicale du collet (la zone rétrécie avec laquelle l'anévrisme communique avec l'artère). En effet, le collet que l'on voit sur la figure est plutôt la paroi du vaisseau qui existerait s'il n y avait pas un anévrisme. La figure 4.2 illustre cette différence.

FIG. 4.2 – Différence entre les deux définitions

Mais, en même temps on peut profiter de notre estimation pour détecter le collet, c'est à dire, parmi les voxels de cette estimation, trouver ceux qui forment une courbe fermée. Cette courbe va nous servir comme une initialisation pour détecter le collet. Il y a donc deux étapes : premièrement, trouver une courbe fermée, deuxièmement, optimiser cette courbe pour qu'elle soit la plus courte possible. Ces étapes sont détaillées dans les deux sections suivantes.

4.2 Deuxième estimation : courbe fermée

La première étape pour trouver le collet tel qu'il est défini médicalement, est de trouver une courbe fermée parmi les voxels de la première estimation du collet (voir section 4.1).

L'algorithme que je propose est basé sur l'algorithme de Dijkstra. Tout d'abord, je veux préciser qu'on va travailler sur les voxels de la surface de la première estimation du collet, je vais appeler ce groupe de voxels *Surface1* dans la suite. L'algorithme

est le suivant :

1. Construire l'arbre de Dijkstra dans *Surface1* en prenant n'importe quel voxel comme source.
2. Chercher le voxel le plus loin de la source *E1*, stocker les voxels de la branche de l'arbre qui part de *E1* jusqu'à la source dans une liste.
3. Mettre la distance à la source des voxels de la liste et tous leurs descendants à 0.
4. Chercher le voxel le plus loin de la source *E2*.
5. Si *E1* n'est pas un voisin de *E2* :
   - Vider la liste.
   - Chercher le voxel *E1* dont la distance à la source est nulle (de la première moitié) dans le voisinage de *E2*.
   - stocker les voxels de la branche de l'arbre qui part de *E1* jusqu'à la source dans la liste.
6. Ajouter à la liste les voxels de la branche qui part de *E2* vers la source.
7. Les voxels de la liste forment une courbe fermée.

La figure 4.3 illustre le résultat.

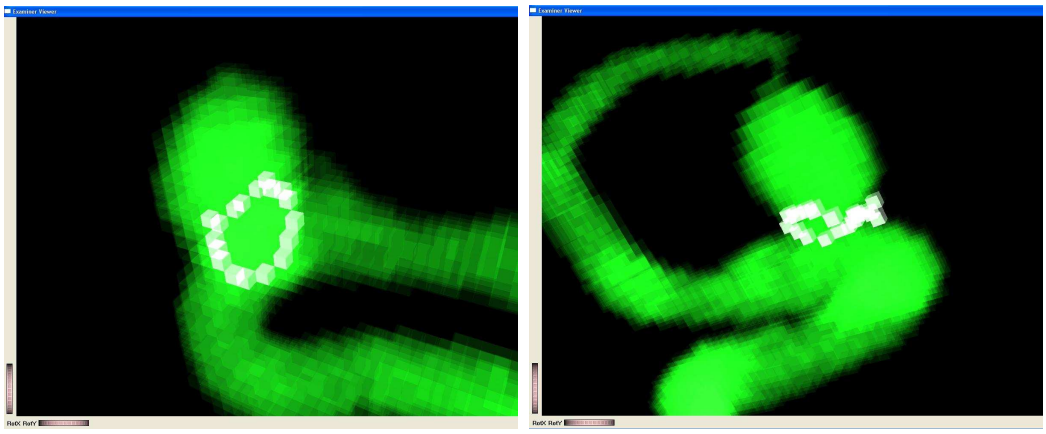


FIG. 4.3 – Courbe initiale

En effet, la définition de l'arbre de Dijkstra (l'arbre des plus courts chemins à un point source), et le fait qu'on construit cet arbre pour un ensemble de voxels

qui forment une surface fermée, garantit qu'il y aura deux "relativement longues" branches de l'arbre qui forment une courbe fermée. La troisième étape garantit que la deuxième extrémité trouvée est dans l'autre moitié de l'arbre, et la cinquième étape garantit la connexion entre les deux branches.

4.3 Optimisation

Après avoir trouvé la liste des voxels de la courbe initiale, il suffit d'appliquer une optimisation pour trouver le collet. L'algorithme d'optimisation que je propose est le suivant :

- Commencer par n'importe quel voxel V_i de la liste, et garder ses deux successeur dans la courbe V_{i+1}, V_{i+2}.
- Si V_{i+2} est un voisin de V_i, supprimer V_{i+1} de la liste. Retourner au début de la boucle.
- Chercher dans le voisinage commun entre V_i et de V_{i+2} (les voisins qui sont de la surface du vaisseau), un voxel $V`_{i+1}$ tel que :
  $Dist(V_i, V`_{i+1}) + Dist(V`_{i+1}, V_{i+2}) < Dist(V_i, V_{i+1} + Dist(V_{i+1}, V_{i+2})$
- Si il existe un tel voxel $V`_{i+1}$, mettre : $V_{i+1} = V`_{i+1}$
- Avancer : $i = i + 1$
- Répéter jusqu'à avoir parcouru complètement une fois l'ensemble de la courbe sans modification.

Sur la figure 4.4 on constate le résultat de cet algorithme.

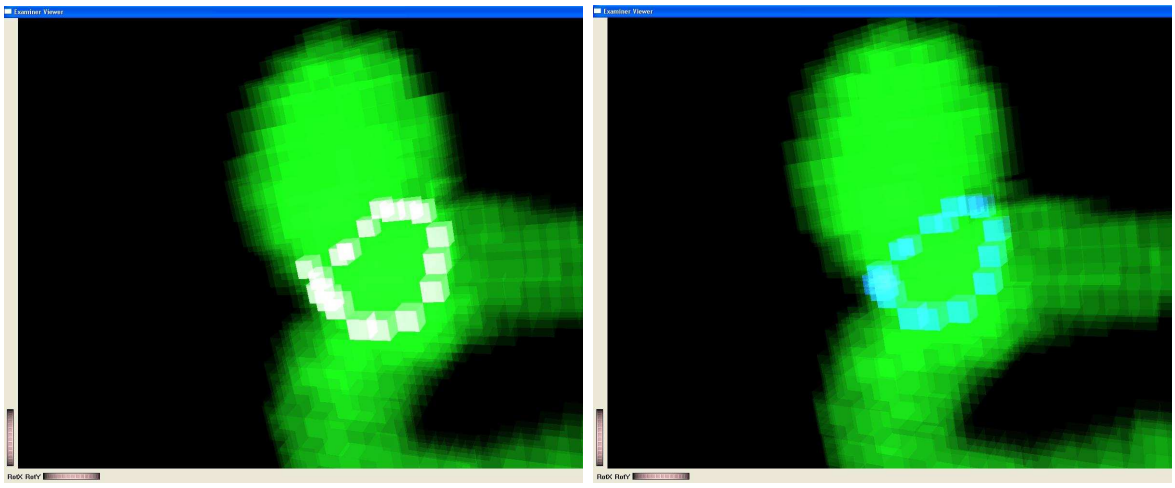


FIG. 4.4 – Le collet : avant l'optimisation (à gauche),après(à droite)

Malheureusement, par manque de temps, je n'ai pas pu assez tester cet algorithme. Mais, il reste prometteur théoriquement et selon cette première implementation.

CHAPITRE 5

Conclusion et Travaux Futurs

Dans ce rapport, j'ai présenté une méthode qui, à partir d'un groupe d'images DICOM qui représentent des coupes dans le cerveau, construit un modèle 3D qui représente l'arbre vasculaire et le visualise.

J'ai également proposé un algorithme pour extraire le squelette de l'ensemble de voxels. Cet algorithme garantit que le squelette soit le plus au centre, connexe et singulier, de plus il garantit que les jonctions entre les branches du squelette soient les plus naturelles possible.

Puis, en me servant de ce squelette, j'ai proposé une méthode pour détecter les anévrismes et leurs collets d'une façon totalement automatique.

Cependant, j'ai donné à l'utilisateur la possibilité d'intervenir où je l'ai vu utile, par exemple, il peut choisir une zone d'intérêt, changer le seuil utilisé pour la segmentation, et choisir de visualiser ou pas un certain groupe de voxels.

Il reste du travail à accomplir. D'abord, il me faut faire plus de tests surtout au niveau de la détection automatique des anévrismes et des collets. Puis, je vais améliorer ma méthode de détection des collets, surtout l'étape d'optimisation, pour la rendre plus robuste.

Ensuite, je compte me servir du collet de l'anévrisme pour déterminer sa taille exacte. Car, une fois le collet précisément détecté, on peut considérer tous les voxels qui sont au dessus du collet une représentation exacte de l'anévrisme.

Enfin, dans le futur, j'envisage de continuer à travailler sur le sujet des anévrismes, dans le sens de simuler le processus d'embolisation. Une telle simulation pourra aider le médecin à estimer le résultat de la chirurgie.

BIBLIOGRAPHIE

[1] Aly A. Farag, Hossam Hassan, Robert Falk, and Stephen G. Hushek. 3D volume segmentation of MRA data sets using level sets : Image processing and display. *Academic Radiology*, 11(4) :419–435, 2004.

[2] N. Flasque, M. Desvignes, J-M Constans, and M. Revenu. Acquisition segmentation and tracking of the cerebral vascular tree on 3D magnetic resonance angiography. *Medical Image Analysis*, 5(3) :173–183, 2001.

[3] Gyi, Erich Sorantin, Emese Balogh, Attila Kuba, Csongor Halmai, Bal&#225 ;zs Erdohelyi, and Klaus Hausegger. A sequential 3D thinning algorithm and its medical applications. In *IPMI '01 : Proceedings of the 17th International Conference on Information Processing in Medical Imaging*, pages 409–415, London, UK, 2001. Springer-Verlag.

[4] Guangxiang Jiang and Lixu Gu. An automatic and fast centerline extraction algorithm for virtual colonoscopy. *Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005. 27th Annual International Conference of the*, pages 5149–5152, 2005.

[5] Yuan Jin and Hanif M. Ladak. Software for interactive segmentation of the carotid artery from 3D black blood magnetic resonance images. *Computer Methods and Programs in Biomedicine*, 2004.

[6] R. Kimmel and J. Sethian. Computing geodesic paths on manifolds. *National Academy of Sciences*, 95(15) :8431–8435, 1998.

[7] K. Krissian, C.F. Westin, and R. Kikinis. Fast and automatic vessel centerline detection for MRA. 2002.

[8] Christophe Lohou and Gilles Bertrand. A new 3D 6-subiteration thinning algorithm based on p-simple points. In *DGCI '02 : Proceedings of the 10th International Conference on Discrete Geometry for Computer Imagery*, pages 102–113, London, UK, 2002. Springer-Verlag.

[9] William E. Lorensen and Harvey E. Cline. Marching cubes : A high resolution 3d surface construction algorithm. In *SIGGRAPH '87 : Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACMPress.

[10] Suhuai Luo and Jesse S. Jin. Recent progresses on cerebral vasculature segmentation for 3D quantification and visualization of MRA. In *ICITA '05 : Proceedings of the Third International Conference on Information Technology and Applications (ICITA'05) Volume 2*, pages 656–661, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Nicolas Passat, Christian Ronse, Joseph Baruthio, Jean-Paul Armspach, and Claude Maillot. Magnetic resonance angiography : From anatomical knowledge modeling to vessel segmentation. *Medical Image Analysis*, 10(2) :259–274, April 2006.

[12] Ming Wan, Zhengrong Liang, Qi Ke, Lichan Hong, Ingmar Bitter, and Arie E. Kaufman. Automatic centerline extraction for virtual colonoscopy. *IEEE Trans. Med. Imaging*, 21(11) :1450–1460, 2002.

[13] Jaeyoun Yi and Jong Beom Ra. A locally adaptive region growing algorithm for vascular segmentation. *International Journal of Imaging Systems and Technology*, 13(4) :208–214, 2003.

ANNEXE A

Segmentation et Voxélisation

A.1 Segmentation

Mes données sont des images en 2D :

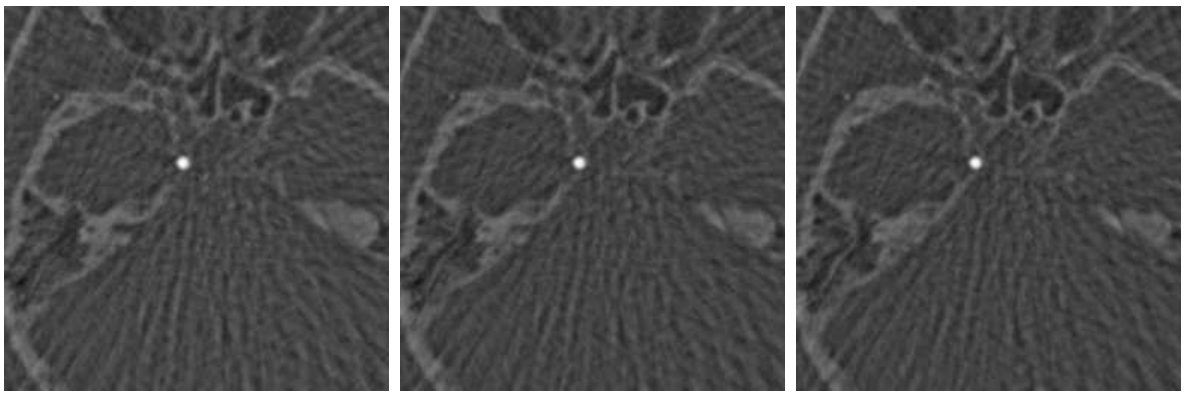


FIG. A.1 – Images DICOM avant Segmentation

En effet, j'ai 150-200 images (256*256 pixels) en niveaux de gris, chaque une présente une coupe longitudinale du cerveau. Le format de ces fichiers est DICOM (Digital Imaging and COmmunications in Medecine) qui est un standard de communication et d'archivage en imagerie médicale, et qui aussi par extension le format de fichier faisant référence dans le domaine de l'imagerie médicale. Les petits cercles blancs que l'on voit sur la figure A.1 sont des coupes des vaisseaux sanguins. Les autres choses présentes dans les images sont les os, les tissues et du bruit. Donc, il

faut faire la **segmentation** pour supprimer toutes les choses autres que les cercles blancs.

Comme mes données sont en 2D, et je dois faire la visualisation en 3D, il fallait répondre à la question : Doit-on faire la segmentation avant de construire la voxélisation ou après ?

A.1.1 Segmentation en 2D

La première idée était de faire la segmentation en 2D, parce que la plupart des méthodes de segmentation existantes :*Croissance de Régions, Segmentation Basée sur Watersheds, Segmentation basée sur Lignes de Niveaux, Méthodes Hybrides, Extraction de Caractéristiques*, sont applicables sur des données en 2D, et sont implémentées dans la librairie ITK que j'ai téléchargé. J'ai essayé quelques un de ces algorithmes codés. Par exemple, pour utiliser le *Connected threshold Image filter*, l'algorithme demande quatre paramètres : *coordonnées du Point Source(x, y), le seuil minimum et le seuil maximum.*

Mais le problème était de trouver les bons paramètres, et avec 150-200 images, ce n'est pas très pratique. C'était pareil pour les autres solutions que j'ai essayés.

Puis, transformer les images résultantes en un fichier 3D, n'é tait pas évident non plus. Comme je vais l'expliquer dans la section suivante.

A.1.2 Segmentation en 3D

L'autre option était de faire un fichier 3D avant faire la segmentation. Ceci était possible avec ITK, dans lequel il existe un algorithme qui transforme un série de DICOM en un fichier 3D sous plusieurs formats possibles (*.analyse, *.vtk, *.Gipl, *.nrrd, ...etc).

Le problème, à ce point là, était de segmenter le fichier 3D. Il n'existe pas d'algorithmes qui peuvent acceder les fichiers 3D ainsi construits.

En fait, j'ai essayé d'écrire un code moi même, mais les formats étaient trop complexes à apprendre en un temps limité.

A.1.3 La Solution

Finalement, la solution retenue fut de transformer chaque image DICOM en un format plus simple, puis d'appliquer une méthode simple de segmentation sur cette image.

Le format que j'ai choisi est le format pgm non compressé.

Pour transformer les fichier que j'ai en .pgm, j'ai utilisé l'utilitaire *Image Magik*. La commande (-convert) permet de transformer le format d'une image très simplement. On peut aussi avec cette commande, en utilisant l'option (-threshold), faire une segmentation très simple qui demande un seul paramètre qui est le seuil. Donc, j'ai fait cette segmentation, en utilisant le même seuil pour toutes les images, et les résultats étaient très concluants, comme on peut voir sur la figure au dessous :

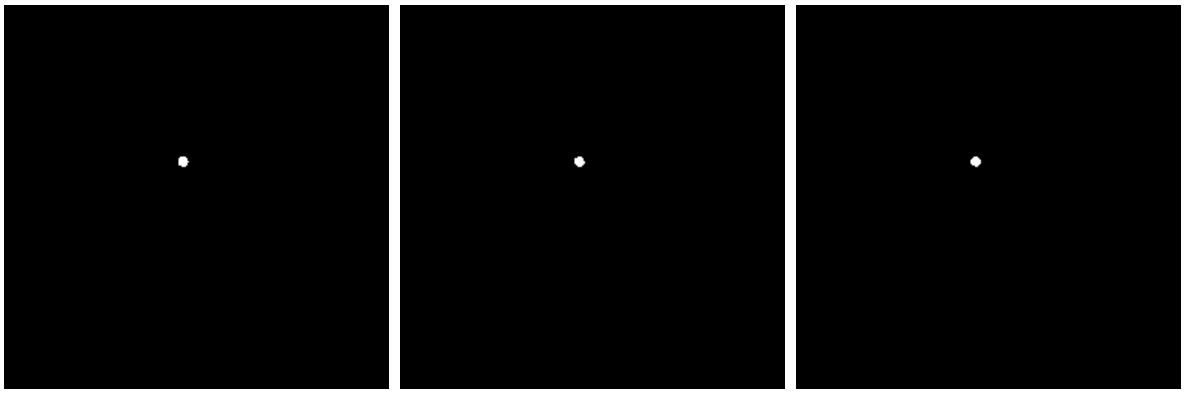


FIG. A.2 – Images de la figure A.1 après la Segmentation

On peut voir très bien les cercles blancs que l'on voulait avoir après la segmentation, et correspondants à la coupe d'un vaisseaux sanguin.

Le seuil peut être choisi simplement, et essayer plusieurs seuils jusqu'a trouver le bon ne prends pas beaucoup de temps, car la segmentation est assez rapide.

A.2 Voxélisation

La voxelisation consiste tout simplement à passer de la 2D à la 3D, c'est à dire à transformer les pixels en voxels. Autrement dit, c'est la construction des cubes, chaque un representant un pixel. Dans mon cas, il faut transformer seulement les

pixels qui appartiennent aux vaisseaux sanguins, c'est à dire les pixels blancs dans les images de figure A.2. Donc, tout ce qu'il faut faire, c'est lire les fichiers (les images), et construire des cubes à partir des pixels blancs.

Les pixel sont en 2D, et les voxels sont des cubes en 3D. Donc, la question qui se pose est d'où vient la troisième dimension ?.La réponse de cette question est détaillée dans la section *Comment construire un voxel ?*

A.2.1 Segmentation et Voxélisation En Même Temps

Pour construire la voxelisation il faut parcourir les pixels et tester la valeur de niveau de gris pour chaque pixel pour savoir s'il est blanc (et donc, à voxeliser) ou pas, donc j'ai pensé qu'on peut faire la segmentation en même temps que la construction de nos voxels.

Donc, au lieu de faire la segmentation avant la voxelisation, on fait les deux en même temps. Ainsi, on va tester la valeur de niveau de gris des pixels de chaque images par rapport à un seuil donné par l'utilisateur, si cette valeur est supérieure au seuil, on prend ce pixel et on fait un voxel. Si non, on fait rien, et le pixel n'est pas transformé en voxel.

A.2.2 Essayer Plusieurs Seuils

J'ai noté que la lecture des images prends beaucoup de temps, ce qui est un problème si l'utilisateur veut essayer plusieurs seuils sur le même group d'image. Donc, et au lieu de relire les image pour chaque seuil, on lit les images dans un tableau où on garde pour chaque pixel sa position 3D (x, y, le numéro de fichier), et sa valeur de niveau de gris.

Puis, on demande à l'utilisateur son seuil, et on accède ce tableau pour faire les étapes suivantes. Si, l'utilisateur veut essayer un autre seuil, on ne lit plus les images, mais on accède à notre tableau des pixels.

A.2.3 Comment Construire Un Voxel ?

Comme on l'a déjà dit, la construction d'un voxel à partir d'un pixel, correspond à la construction d'un cube. Pour définir un cube on a besoin soit d'un sommet et de la longueur du côté, soit des huit sommets. Ici, on a choisi de définir les huit sommets du cube, à cause du viewer utilisé. Je vais plus expliquer ça dans le chapitre suivant.

Le passage d'un pixel au voxel correspondant se fait de la manière suivante :

Dans une image n, on dit que les coordonnées 3D d'un pixel sont :

- x qui correspond à la place du pixel par rapport au largeur de l'image
- y qui correspond à la place du pixel par rapport à l'hauteur de l'image
- z qui correspond au numéro de l'image parmi toutes les images, autrement dit, au niveau de la coupe longitudinale dans le cerveau

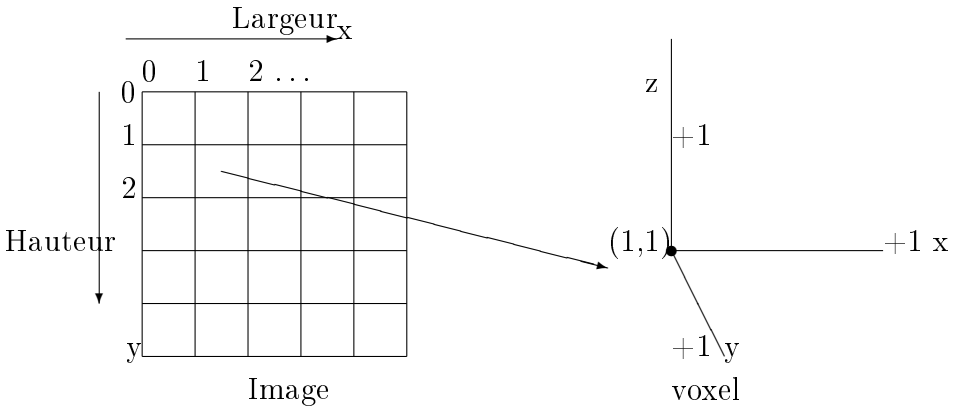


FIG. A.3 – La Construction D'un Voxel

Alors, à partir de ces coordonnées d'un pixel, on extrait les coordonnées 3D des huit sommets du voxel qui correspondent à ce pixel. Le principe était de considerer que les coordonnées du pixels comme les coordonnées d'un sommet, et que la longueur du côté est 1. On a choisi l'entier 1, parce que on parcourt les pixels dans une image, voir la figure A.3, par rapport à la largeur et à la hauteur de l'image, en ajoutant 1 a chaque fois.

Concernant le Z, ou l'hauteur du voxel, elle est également de 1, on a ainsi un cube. De plus le z corresponde ainsi au numéro de chaque image.

Finalement, comme on voit bien sur la figure A.3, les directions des axes (x, y, z), ne correspondent pas à la règle de *right handed*. Pour résoudre ça, il suffit de multiplier toutes les valeurs de (y) par -1.

ANNEXE B

Visualisation

Open Inventor est une interface qui se trouve dans la *Coin3D*, qui est une boîte à outils graphiques haut-niveau pour développer un logiciel de visualisation 3D.

B.1 Pourquoi Open Inventor ?

On a choisi d'utiliser Open Inventor pour la visualisation inspiré par le travail fait par Dr.Olivier Palombi. Elle possède aussi de nombreux avantages :

- Cette interface ne requit que la *Coin3D*.
- Quand on a cette *toolkit* installée, il nous suffit d'inclure tous les fichiers en tête nécessaires pour notre program C++.
- En plus, avec Open Inventor, beaucoup de choses se contrôlent simplement, soit dans le programme, soit par l'utilisateur.
- On voit sur la figure B.1 une interface réalisée par Open Inventor, et affichante une partie de l'arbre vasculaire.

On voit sur la figure B.1 que l'utilisateur a beaucoup de choix par rapport à la façon d'affichage des voxels *as is, hidden line, wireframe ...etc.* Et bien-sûr il peut zoomer, tourner les voxels. Il peut aussi contrôler la transparence des voxels.

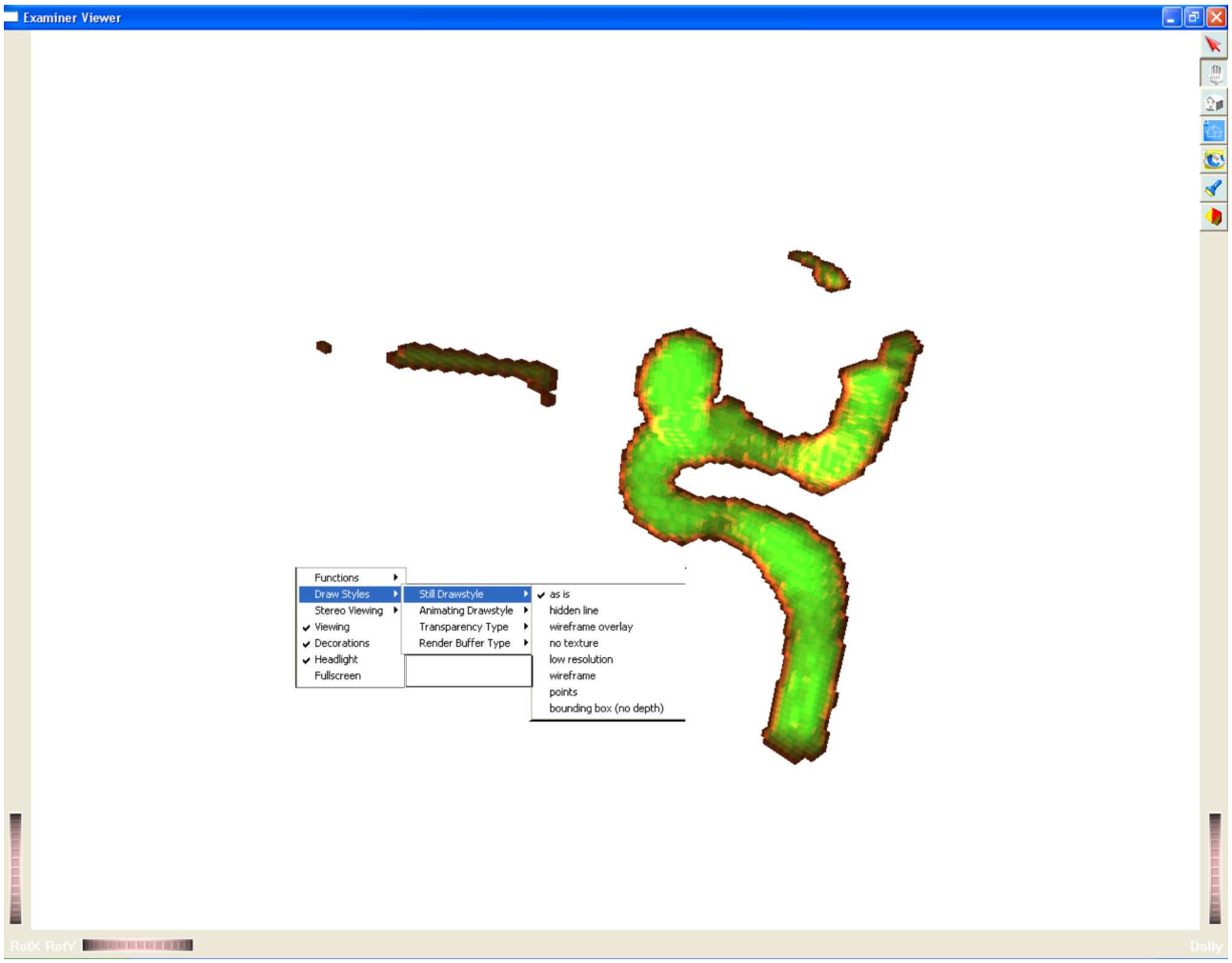


FIG. B.1 – Open Inventor Interface

B.2 Comment Visualiser avec Open Inventor ?

La scène dans Open Inventor est construit d'un noeud de racine et d'autre noeuds fils. Chaque un de ces noeud présente une chose dans la scène (les matériaux, les sommets, les faces, ...etc).

Open Inventor fournit plusieurs méthodes pour visualiser un group de cubes, mais il a toujours besoin des coordonnées 3D de tous les sommets. C'est pour cette raison qu'on a choisi de définir les voxels par leurs huit sommets, comme j'ai expliqué dans le chapitre précédent.

Il existe alors deux méthodes, ou plus précisément, deux noeuds pour faire des cubes : SoFaceSet, SoIndexedFaceSet. Les deux noeuds donnent des faces qui vont construire nos voxels, et les deux ont besoin d'un tableau des coordonnées 3D de tous les sommets.

La difference entre les deux est que avec SoFaceSet, on définit les faces par le nombres de sommet dans chaque face et les normales à chaque face.
Mais avec SoIndexedFaceSet, on a besoin d'un tableau d'indices pour définir quels sommets il faut prendre pour faire les faces. Dans le tableau des indices on sait où la face finit, en utilisant l'indice : *SO_END_FACE_INDEX*.

J'ai choisi de faire mes cubes avec SoIndexedFaceSet, pour trois raisons :

- On n'a pas besoin d'un tableau pour les normales aux faces, ce qui diminue beaucoup la mémoire nécessaire surtout si on se rappelle que on a milliers de voxels, donc milliers de faces.
- On n'a pas besoin de répéter les sommets communs entre les cubes dans le tableau de sommets, chaque sommet est mentionné une seule fois.
- Si on veut faire un autre group de voxels (les voxels de surface par exemple), on peut utiliser les mêmes sommets, et tout ce qu'il faut ajouter, c'est le tableau des indices pour le nouveau group

Donc, jusqu'à maintenant on sait que on a besoin de deux noeuds qui sont : un noeud des coordonnées des sommets, un noeud des indices. Puis, pour définir les couleurs on a besoin d'un noeud de matériau, avant les autres deux noeuds.

B.3 Choix d'Une Zone d'Intérêt

Grâce à open inventor, donner l'utilisateur la possibilité de choisir une zone d'intérêt ne m'a coûté qu'ajouter quelques noeuds.

Le principe choisi est de couper la scène en utilisant six plans, deux pour chaque axe. Pour chaque axe, les valeurs defaults pour les deux plans, sont le minimum et le maximum du coordonnées des sommets sur cette axe. Puis, l'utilisateur peut bouger ces plans en supprimant tous les voxels derrière les plans.

Donc, on a besoin d'un noeud de *SoClipPlane* pour chaque plans. Puis, on met la valeur de ce plan (comme on l'a déjà dit, soit le maximum ou le minimum sur chaque axe). Il ne faut pas oublier de donner au noeud une valeur *true*, pour que la coupe s'effectue.

Puis on a besoin d'un noeud de *SoEventCallBack*, qui nous permet de détecter les mouvement de la souris, les touches du clavier tapées ...etc. On peut, après, préciser quel sont les événements que l'on veut détecter, et quelle est la fonction qu'il faut appliquer quand ces événements s'effectuent.

Pour nous, on a décidé que l'utilisateur doit utiliser le clavier pour bouger les plans. Donc, on a ajouté au noeud de *SoEventCallBack* le *SoKeyboardEvent* en reliant cet événement à une fonction que on a crée. Cette fonction doit détecter quelle est la touch qui était tapée, et réagir par rapport à la touche. Par exemple, on a décidé que pour bouger le plan qui correspond au minimum sur l'axe Z, il faut utiliser les touches *page up*, *page down*. Donc, quand la fonction détecte ces touches, elle va ajouter/sutracter la valeur 1 à la valeur du plan correspondant à Z.

Dans les figures B.2, B.2 on peut voir l'arbre avant et après la selection de la zone d'intérêt.

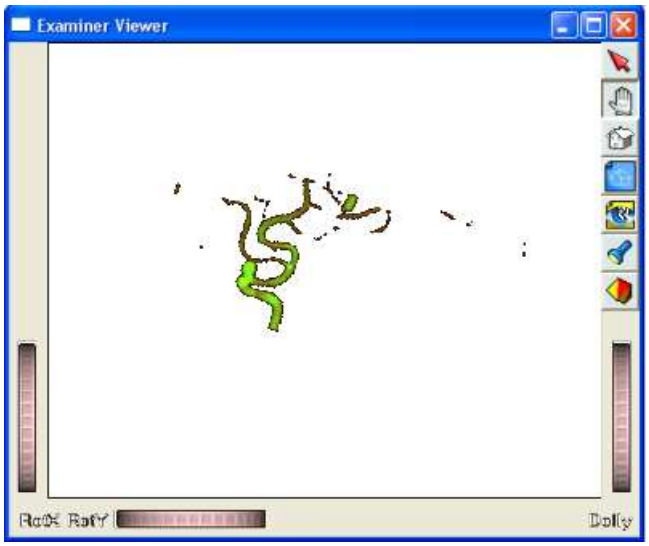


FIG. B.2 – Avant la Sélection

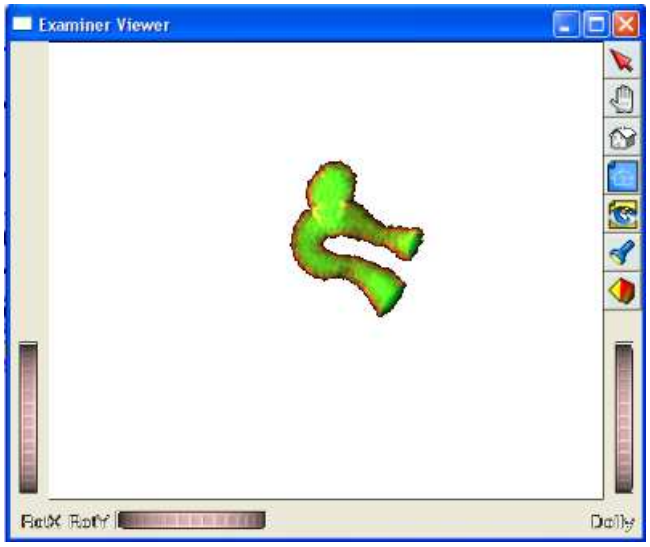


FIG. B.3 – Après la Sélection

On peut se demander : pourquoi sélectionner une zone d'intérêt ? En fait, on a pensé de faire ça pour les raisons suivantes :

- Réduction de la taille de voxélisation, qui économise de la mémoire et du temps
- Simplification de l'arbre vasculaire(on peut supposer que la région sélectionnée correspondra à une seule branche)
- La selection d'une seule composante connexe, alors que la segmentation peut

avoir décomposé l'arbre vasculaire en plusieurs composantes (voir la figure B.2).

B.4 Sélectionner les Choses à Voir

Dans la scène que l'utilisateur voit, il existe quatre group de voxels :

- Les voxels principales c'est à dire tous les voxels construits
- Les voxels de la surface
- Les voxels de la ligne du centre
- Les voxels des anévrismes
- Les lignes qui présentent l'arbre construite comme expliqué dans le dernier chapitre

Parfois l'utilisateur n'est pas intéressé de voir tous les quatre group, pour cette raison, on a associé à chaque group une lettre : (V pour tous les voxels, S pour les voxels de la surface, C pour les voxels de la ligne du centre, et A pour les voxels des anévrismes), puis, en appuyant sur une de ces lettres, l'utilisateur peut visualiser/pas visualiser le group des voxels correspondant. C'était fait grâce au noeud *SoSwitch*.