Réalisation d'opérations morphologiques sur des modèles 3D grâce à OpenVDB

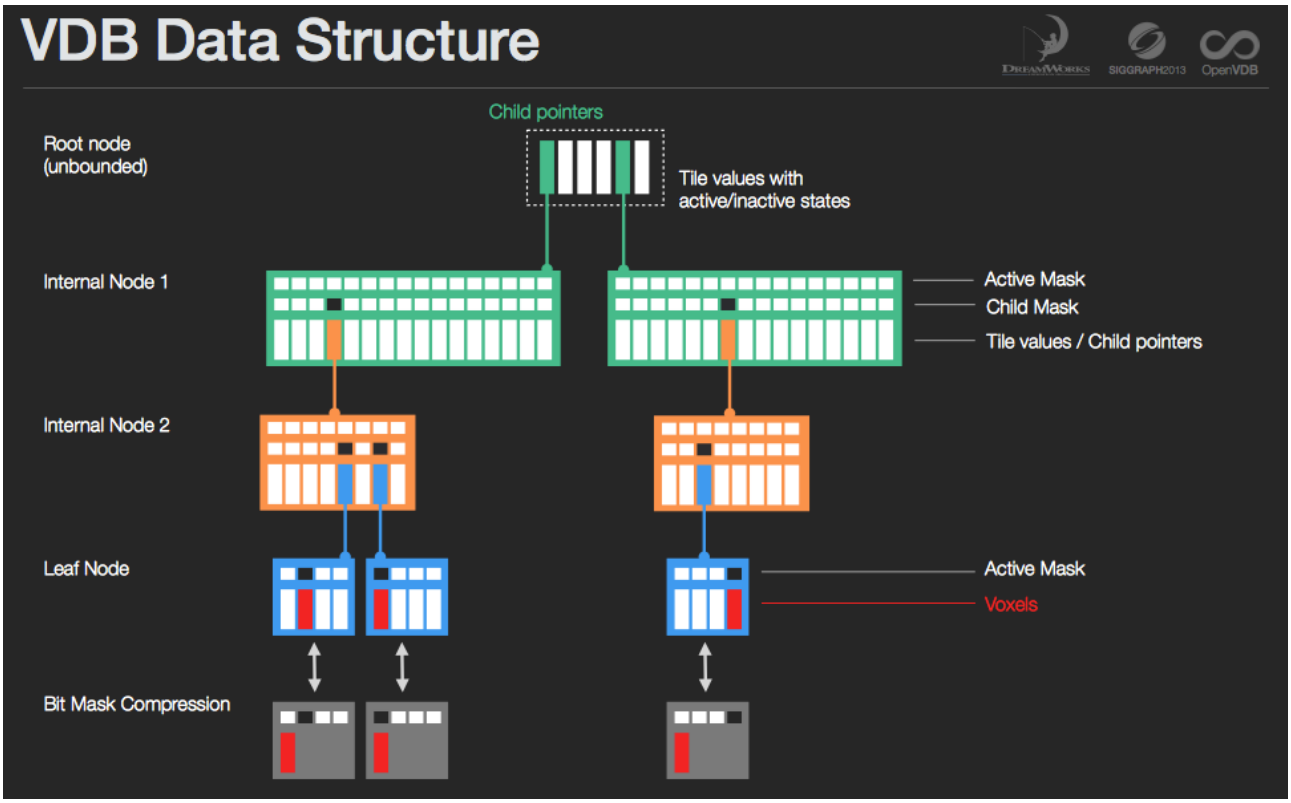
Paola Vaulet
Encadrants : Stéphane Calderon, Tamy Boubekeur

Objectif du projet :

Pouvoir réaliser des opérations morphologiques rapidement sur des modèles 3D grâce à une nouvelle structure de données fournie par OpenVDB. Le projet initial devait comporter une interface graphique pour la rendre plus facile d'utilisation, je reviendrai plus loin sur les raisons qui m'ont empêchée de réaliser cette partie.

OpenVDB :

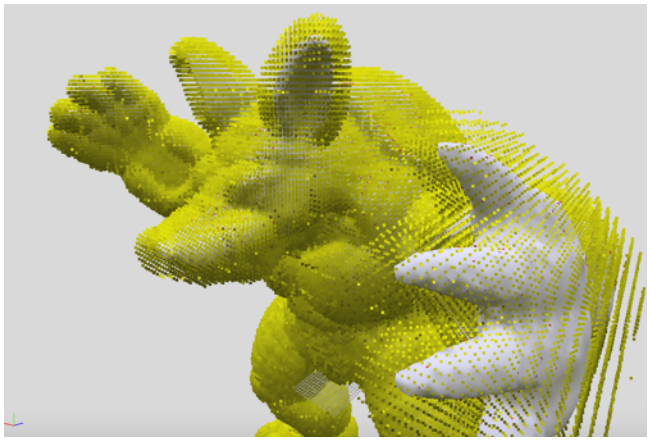
La force d'OpenVDB réside dans les outils mis à la disposition de l'utilisateur pour stocker et accéder aux données. Je vais ici m'attarder sur les *trees*.



Comme tous les arbres classiques, les trees d'OpenVDB possèdent une racine, un ou plusieurs noeuds internes et des feuilles qui contiennent les données. Toutes les feuilles se trouvent à la même profondeur. Chaque noeud interne contient un nombre N de fils, N pouvant s'écrire $(2^p)^3$. Les feuilles de l'arbre contiennent finalement les voxels qui, eux, peuvent contenir à peu près ce que l'on veut (le plus classique étant des *float* ou des *int*). Ce qu'il est aussi intéressant de noter sur ce schéma est la notion d'activation ou non des noeuds internes, des feuilles, ainsi que des voxels. Un voxel n'est activé que s'il est intéressant (notion à définir par l'utilisateur). Une feuille n'est active que si elle contient un voxel actif et ainsi de suite pour les noeuds internes. C'est ce qui va permettre de considérablement accélérer les accès mémoire puisque pour chaque opération, il suffit de regarder les voxels actifs.

Il est de la responsabilité de l'utilisateur de définir la profondeur de l'arbre, l'arité à chaque niveau et le type de données contenues. Dans mon cas, j'ai utilisé un Tree4<float, 5, 4, 3> donc un arbre de profondeur 4, contenant des *float*, le premier noeud interne contenant jusqu'à 32x32x32 noeuds internes puis jusqu'à 16x16x16 feuilles et celles-ci contenant 8x8x8 voxels.

OpenVDB propose également des outils afin de convertir un *mesh* fermé (un ensemble de points et de polygones) en une courbe de niveau (un *level set*). Les voxels de l'arbre obtenu représentent alors un champ de distance à la courbe. Les voxels positifs sont à l'extérieur de la courbe et les voxels négatifs à l'intérieur.



Exemple de conversion d'un modèle .off en champ de distances (on remarque une mince couche de voxels actifs en jaune)

Algorithme utilisé :

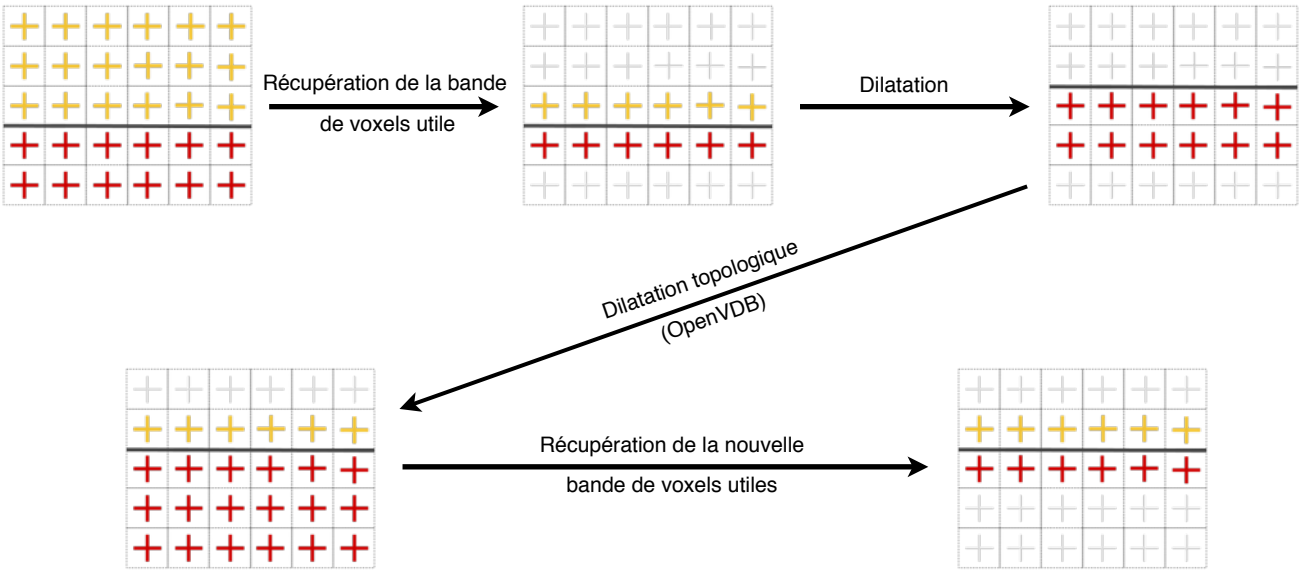
Afin de pouvoir réaliser une opération de morphologie simple (dilatation ou érosion), il faut pouvoir trouver la surface du modèle, c'est à dire les voxels possédant dans leur voisinage un voxel de l'autre côté de la surface, donc de signe opposé. Un premier passage sur tous les voxels activés permet de ne garder qu'une très fine couche de voxels actifs qui serviront ensuite à l'opération morphologique. Pour des raisons de simplicité, dès cette première passe, je ne conserve pas le champ de distance mais mets tous les voxels extérieurs et intérieurs à une valeur ±1.
Ensuite, l'opération de morphologie est classique :

Dilatation : $\forall x \in \mathbb{R}^n,\ D(f,B)(x) = \sup\{f(y) \,/\, y \in B_x\}.$

Erosion : $\forall x \in \mathbb{R}^n,\ E(f,B)(x) = \inf\{f(y) \,/\, y \in B_x\}.$

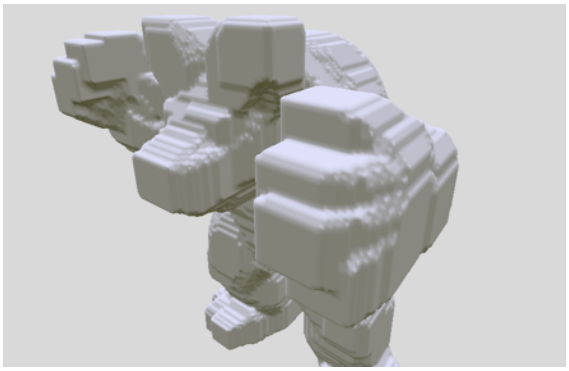
Avec *f* la fonction à dilater (ici la surface de l'objet) et *B* l'élément structurant (ici défini par le voisinage choisi). Dans mon cas, il est possible de réaliser des opérations en utilisant la 6 (faces), 18 (arêtes) ou 26 (coins) connexité.

Une fois ceci réalisé, il faut réaliser une dilatation topologique sur les voxels actifs de l'arbre puis de nouveau récupérer la fine couche de voxels intéressants autour de la surface.

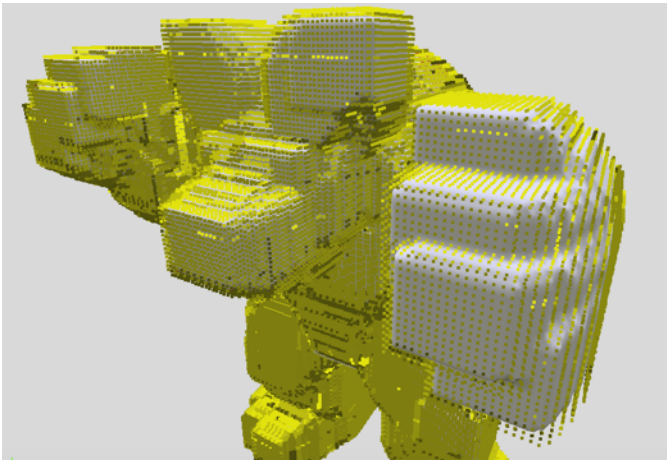


Résultats :

Cette méthode permet d'obtenir les opérations rapidement. Chaque opération se fait en 3-4s environ pour des modèles avec de l'ordre de 10^5 voxels.



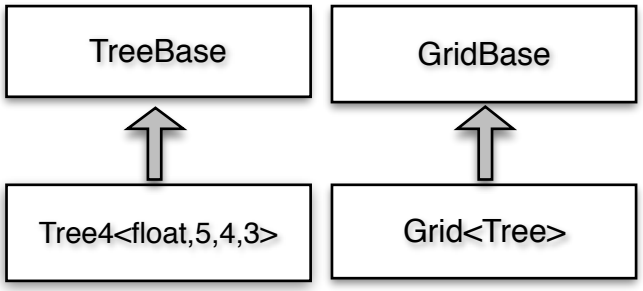
Résultat de 6 dilatations successives (26-connexité) et la couche de voxels actifs



Problèmes rencontrés, limitations et améliorations possibles :

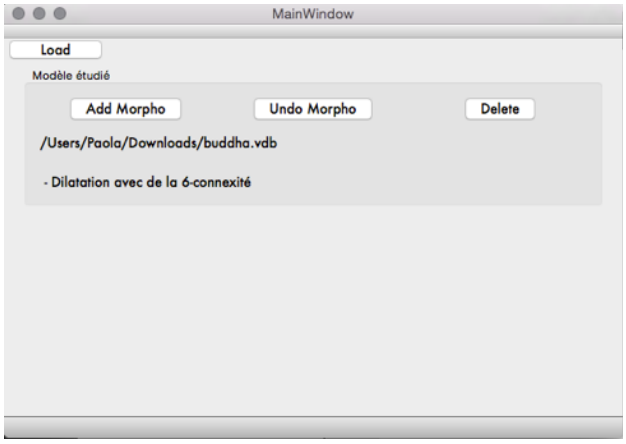
Tout d'abord, n'étant pas familière avec l'installation et l'utilisation de nouvelles librairies en C++, j'ai mis énormément de temps à installer et faire fonctionner OpenVDB sur mon ordinateur… Et la bibliothèque n'a fonctionné qu'avec Xcode.

OpenVDB est un monde très *templaté.* Un monde où toutes les classes instanciables sont *templatées* et héritent d'une classe de base abstraite non *templatée* et ne possédant pas les mêmes attributs que les classes qui en héritent.



C'est par exemple une GridBase qui est prise en paramètre par le *viewer*. Le problème c'est qu'elle renvoie un *TreeBase* lorsqu'on récupère le *Tree* qu'elle contient. Or ce *TreeBase* ne possède pas de noeud interne, ni de feuilles etc… et il n'est pas possible de le caster de manière statique ou dynamique. J'ai donc dû utiliser une variable globale et commune aux différents fichiers contenant le *Tree* sur lequel travailler. Cette méthode n'est vraiment pas élégante mais c'est la seule que j'ai trouvée. Le type d'arbre utilisé est donc défini à l'avance, ce qui n'est pas non plus optimal sauf si on sait à l'avance la taille des données sur lesquelles travailler. Ce *templatage* qui offre une grande liberté a priori sur la structure de l'arbre s'est révélé très contraignant finalement… J'ai par exemple essayé d'implémenter un « *undo* » sur les données. Cela n'a pas non plus été possible car la copie profonde de l'arbre (qui m'aurait permis de conserver une « trace du passé ») ne renvoie qu'un *TreeBase* donc encore un arbre « inutilisable ».

Une amélioration possible et rapide à implémenter serait une interface graphique pour faciliter la prise en main. J'en ai réalisé une avec Qt Creator et utilisant la bibliothèque Qt. Seulement, j'ai été incapable de compiler un quelconque programme utilisant la bibliothèque Qt sur Xcode.



Voilà à quoi aurait pu ressembler l'interface graphique si j'avais pu la compiler avec Xcode

Une autre amélioration possible aurait été d'étudier la possibilité de conserver un champ de distance pour les voxels et ainsi réaliser des opérations morphologiques avec des éléments structurants de forme et taille quelconque.