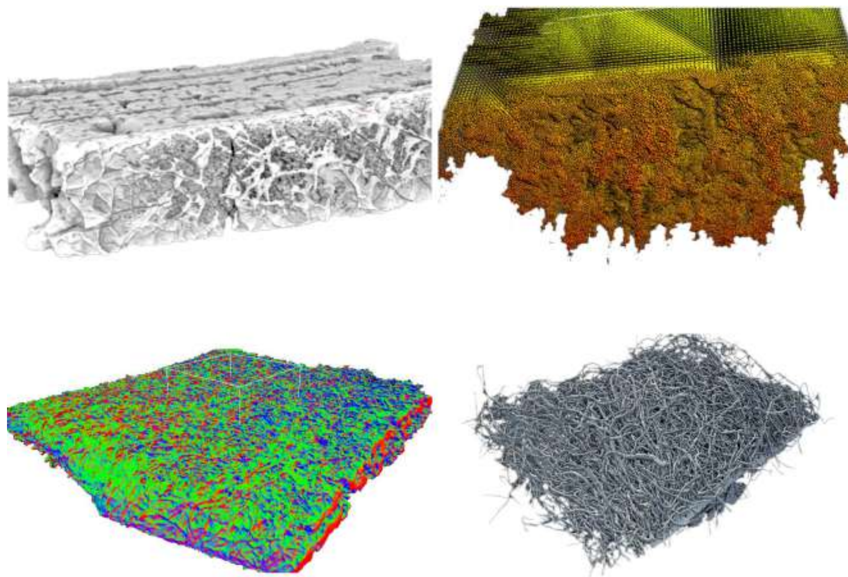
Rapport de stage

Modélisation 3D de volumes à l'échelle nanométrique et la simulation de fluides avec OpenGL/CUDA



1er juin au 31 août

Encadrants :

M. Benoît CRESPIN
M. Philippe MESEURE
M. Maxime MARIA
M. Samuel PELTIER

Stagiaires :

Yoann SOCHAJ
Josué RAAD

xlim

Table des matières

Table des matières 2

Introduction 3

Modélisation et traitement 4

- Les données 4
- Pré-traitement des images 4
- Génération des meshes 6
  - Approche volumique 6
  - Approche surfacique 7
- Traitement des meshes 7
  - Décimation 8
  - Géométries dégénérées 8
  - Composantes connexes 9
  - Billes de cuivre 9
  - Autres traitements 10
  - Limites 10
- Visualisation 11
- Résultats 12
- Perspectives 12
- Guide d’utilisation 13
  - Génération d’un mesh 13
  - Traitement d’un mesh 13
  - Visualisation 14

Simulation 15

- Importation d’objets 3D : 15
- Interaction particules/objets 15
- Optimisations 16
  - Optimisation du temps 16
  - Optimisation de mémoire 18
- Les billes de cuivre 19
- Guide d’utilisation 20
  - Les étapes de la simulation 20
  - Les choses à savoir 21
- Démonstration 22

Introduction

De nos jours, l'hydrogène est un gaz qui peut être utilisé pour générer et stocker de l'énergie. Il faut savoir que le coût pour le créer n'est pas négligeable du point de vue environnemental.

Le but du projet est de pouvoir fabriquer de l'hydrogène à l'aide de céramique et d'une réaction chimique avec de l'acide borique et des dépôts de cuivre dans la céramique.

Pour maximiser le taux d'interaction entre le fluide et les complexes métalliques, il faut que la céramique soit la plus poreuse possible. C'est pourquoi la céramique est imprégnée dans des structures fibreuses. La céramique est ensuite traitée thermiquement pour verrouiller sa forme.

Des acquisitions tomographiques ont été réalisées par l'IRCER, notre travail commence ici.

Notre but est de modéliser des objets 3D à partir de ces acquisitions. Pour ensuite simuler un fluide et déterminer le taux d'interaction de l'acide. À terme, on cherche à savoir précisément quelles sont les configurations optimales pour propager fluide dans la céramique et quelles sont les configurations les plus intéressantes.

Modélisation et traitement

Dans cette partie du projet, le but est de partir des données brutes provenant directement des acquisitions tomographiques[1] pour produire des objets 3D compatibles avec la simulation de fluide. Nous allons explorer les différentes approches et les outils ont été étudiés pour mener cette tâche à bien.

Les données

Les données à traiter se présentent sous la forme d'une séquence d'images obtenues par tomographie. Ces images sont empilées sur l'axe Z et mises bout à bout, permettent de reconstituer le volume de l'objet scanné. L'intensité des pixels de l'image représentent différentes propriétés du matériau ou l'absence de matériau.

On dispose de données représentant soit des structures déjà imprégnées de céramique (traitées thermiquement ou non) comme sur la figure 1. Ou alors de fibres simples non imprégnées comme sur la figue 2.

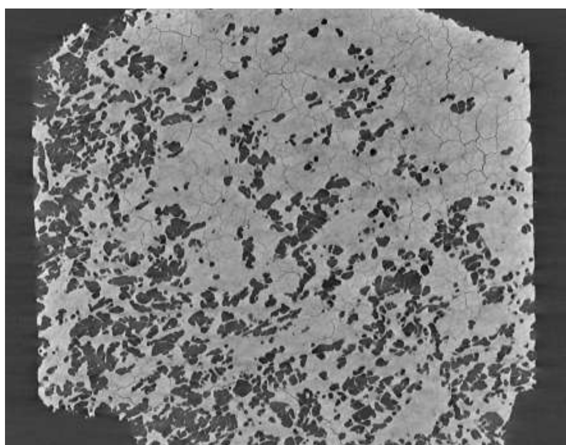


Fig. 1 : Exemple d'une image brute venant de l'acquisition d'une céramique traitée thermiquement.

Fig. 2 : Exemple d'une image brute venant de fibres non imprégnées

Pré-traitement des images

À première vue, les images brutes s'avéraient difficile à traiter correctement pour extraire les fibres. On remarque déjà sur la figure 3 que les parties claires et sombres correspondent toutes deux à une fibre, et qu'il faut extraire le fond gris comme sur la figure 4 . Les valeurs qui correspondent à la partie intéressante du mesh varient d'une acquisition à l'autre. Alors, on utilise une analyse de l'histogramme pour déterminer ces valeurs.

[1] https://en.wikipedia.org/wiki/Tomography

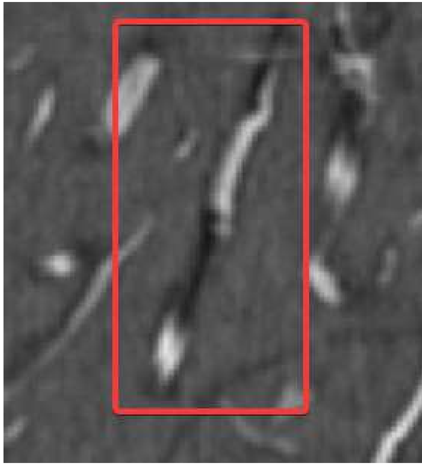


Fig. 3 : Gros plan sur une fibre d'une image originale

Fig. 4 : Gros plan sur la même fibre de l'Image traitée

Un autre souci qui a pris beaucoup plus de temps à régler fut d'extraire les structures fibreuses des données propargilées. Sur la figure 5 , les fibres sont à peine visibles, à force de filtres et traitements avec GIMP dans un premier temps, puis OpenCV, on peut obtenir une séparation comme sur la figure 6. Cela reste encore loin d'une séparation acceptable.

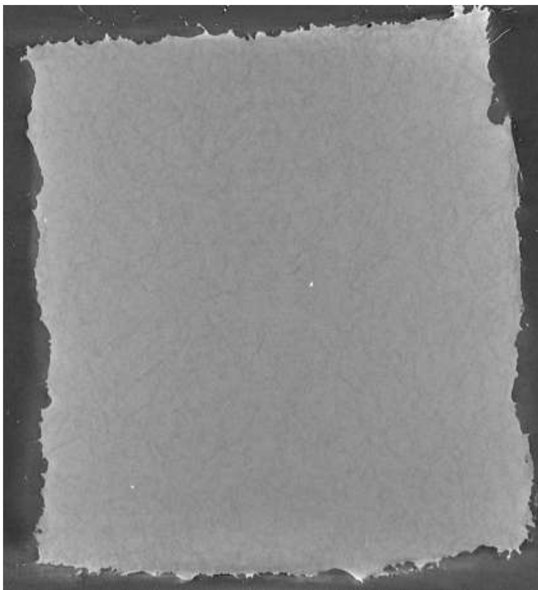


Fig. 5 : Image originale (KPP Prop AHPCS)

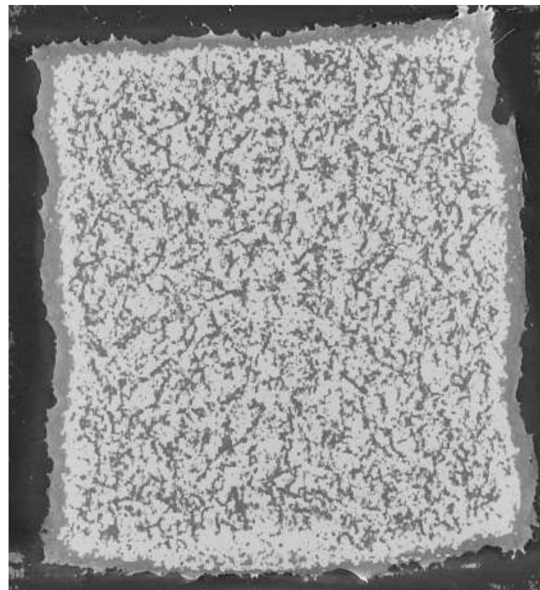


Fig. 6 : Image traitée (KPP Prop AHPCS)

Finalement le passage sur un autre dataset qui permet d'avoir une séparation plus simple, comme sur les figures 7 et 8, débloque la situation.

Ce nouveau traitement permet aussi d'avoir une séparation entre les bordures extérieures et les fibres internes (en rouge et vert sur la figure 9.). Cela aiderait à potentiellement marquer ces zones pour qu'elles puissent bénéficier d'un traitement différent une fois le mesh créé.

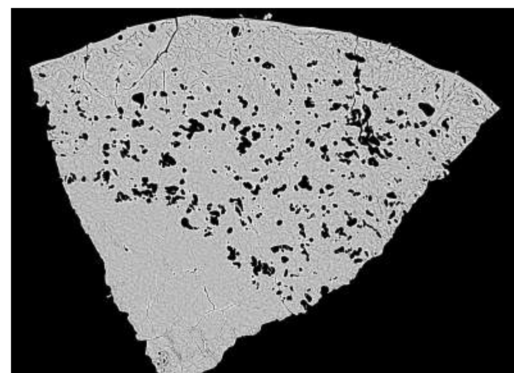


Fig. 7 : Image originale (Dataset 4)

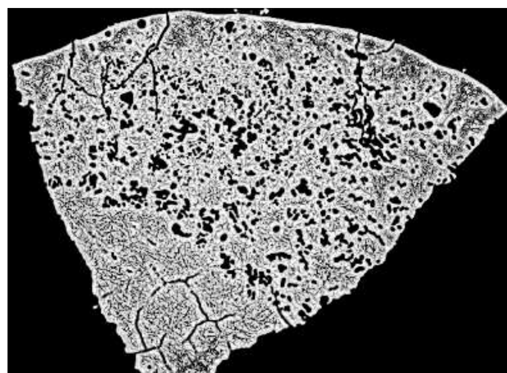


Fig. 8 : Image traitée (Dataset 4)

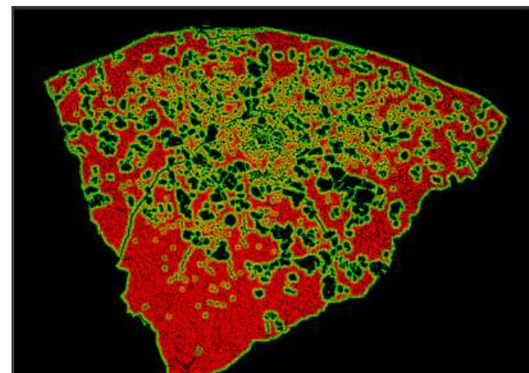


Fig. 9 : Isolation lors du traitement, sur les canaux rouge et vert. (Dataset 4)

Le prétraitement des images a constitué une partie significative de cette partie des travaux, mais finalement, il s'avère que ce n'est pas si nécessaire.

En effet, on a pu établir lors d'une réunion très informative avec Sylvie Foucaud que l'on pouvait avoir des résultats déjà très exploitables sur certains sets de données sans traitement préalable.

Génération des meshes

Pour générer un maillage à partir des séquences d'images, on utilise un algorithme de marching cube. En déterminant une iso-valeur, on détermine une surface au mesh et cela permet d'obtenir un maillage exploitable.

Approche volumique

Dans un premier temps, on utilisait TVMC pour générer non pas une surface, mais un volume. Ce dernier est représenté par une G-Carte[2], ce qui permet de garantir une cohérence sur les meshes créés entre les zones vides et pleines. Cela peut aussi être utile par la suite pour savoir dans si une particule est autorisée ou non à être dans une zone.

Dans notre cas, le désavantage de cette méthode est que la quantité de données est très élevée. Ce qui augmente aussi le temps de génération. En effet, toutes les informations de relations entre les brins ne nous intéressent pas forcément.

Cela a pour conséquence d'augmenter considérablement l'empreinte mémoire. Lorsque toute la mémoire vive est utilisée et l'OS commence à passer sur le swap, la génération devient déraisonnablement lente.

À titre d'exemple, sur 16 GB, on peut traiter environ 15 images de 2 k pixels avant d'utiliser le swap, et approximativement 30 images de même taille avec 32 GB.

Aussi, lorsque plus d'une centaine de Go sont utilisés par TVMC, la library semble ne plus suivre et s'arrêter brutalement. Ce comportement n'a pas été investigué plus que ça, car il n'arrive qu'après plusieurs heures de traitement.

Comme le matériel était limitant, on a traité les meshes progressivement, avec des couches d'une dizaine d'images à la fois. Avec les bons paramètres d'overlap entre ces couches, on obtient un alignement parfait et une combinaison est ensuite possible dans un logiciel externe (Blender). La figure 10 montre ces différentes couches que l'on fusionne.

[2] Très simplement, les G-cartes décrivent les relations entre des éléments simples (les "brins" ou "dart"), et permettent de représenter des structures sur un nombre arbitraire de dimensions. https://en.wikipedia.org/wiki/Combinatorial_map

Finalement, on a réussi à produire des meshes complets à partir d'une approche volumique (CF. Fig 11).

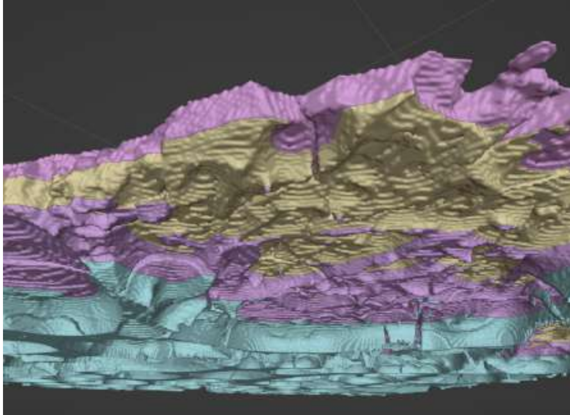


Fig. 10 : Les différentes couches exportées par TVMC (de couleurs distinctes)

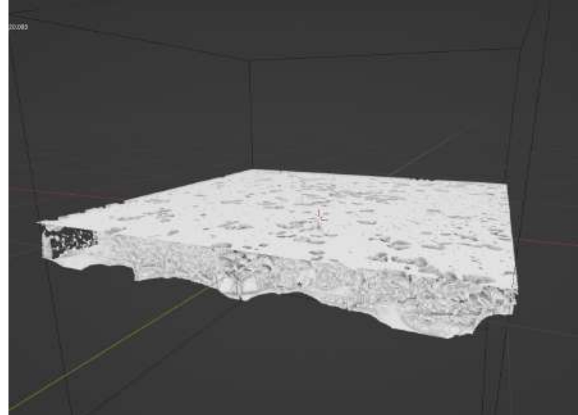


Fig. 11 : Mesh complet construit à partir d'une approche volumique.

Cependant, au cours du projet, la question de l'utilité réelle d'une telle approche s'est posée. Tout compte fait, n'y aurait-il pas trop d'inconvénients à gérer de tels modèles alors que la simulation s'intéresse pour le moment uniquement aux collisions avec la surface du mesh. De plus, les fusions et les traitements du mesh se font sans aucune information volumique.

Ainsi donc, malgré les quelques avantages, comme les informations sur le volume actuel ou les relations d'adjacences, nous sommes passés à une approche totalement surfacique.

Approche surfacique

Après quelques recherches sur le marching cube surfacique, la découverte de la bibliothèque PyMcubes[3] pour python a séduit par sa simplicité d'utilisation, et sa performance (Cpp multithread).

Une simple application python a alors été créée pour produire un marching cube surfacique. Le but était de reproduire les options disponibles dans TVMC, notamment les offsets et le nombre d'images à traiter. Mais également d'ajouter quelques fonctionnalités supplémentaires, comme un padding ou un multiplicateur de résolution.

Le facteur limitant n'est maintenant plus la génération du mesh, mais la vitesse d'écriture de gigantesques fichiers .obj sur le disque.

Désormais, la génération de meshes correspondants même aux datasets les plus grands ne prend que quelques minutes.

Traitement des meshes

Les maillages créés peuvent parfois être composés de plusieurs centaines de millions de triangles. Il serait inapproprié d'utiliser ces meshes tel quel pour la simulation. C'est pourquoi un traitement post génération est nécessaire.

Dans cette partie, le traitement est réalisé sous Blender [4] à partir des fichiers .obj donnés par la génération. Le résultat des opérations est enregistré dans un nouveau fichier .obj.

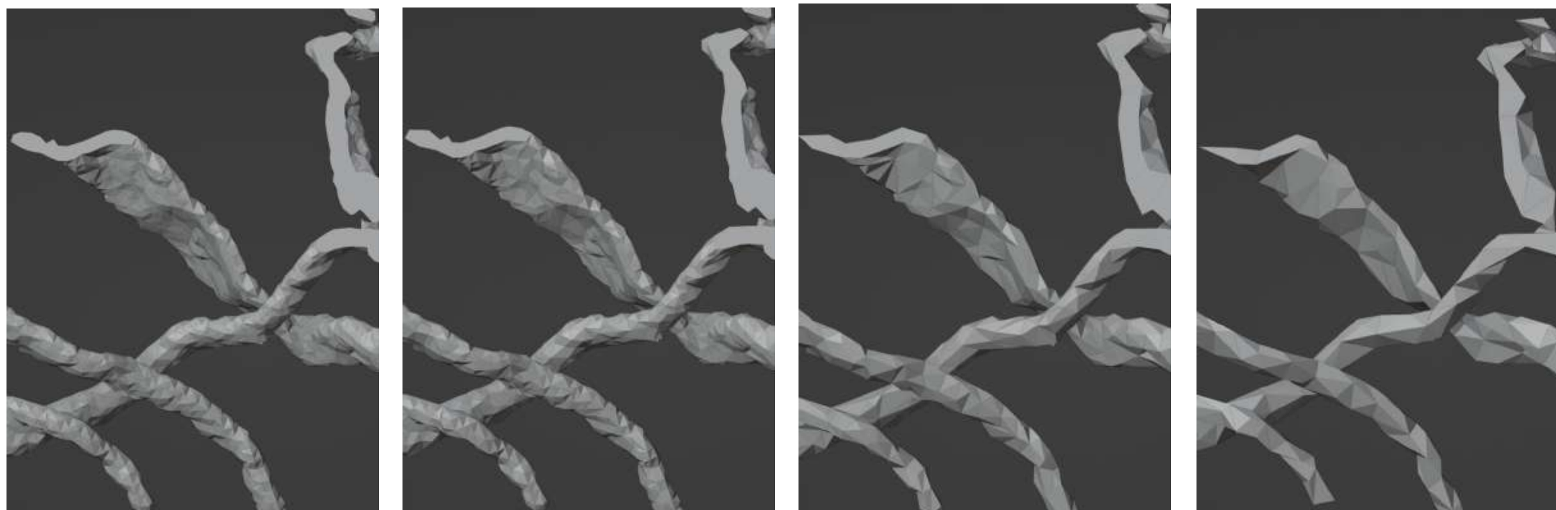
---

[3] Installable avec PIP : https://pypi.org/project/PyMCubes/
[4] https://www.blender.org/

Décimation

Le traitement le plus adapté est la décimation. De toutes les méthodes testées, “Edge collapse” produit toujours les meilleurs résultats. La décimation permet de réduire le nombre de triangles à une fraction du nombre originale. Par exemple, une décimation de 10 % d’un mesh de 150M de triangles donne un mesh de 15M de triangles.

La figure 12 illustre une telle décimation. L'intérêt ici est que l’on garde approximativement le même volume du mesh.



Aucune décimation (9.1 M tri) Décimation 50 % (4.5 M tri) Décimation 20 % (1.8 M tri) Décimation 10 % (0.9 M tri)

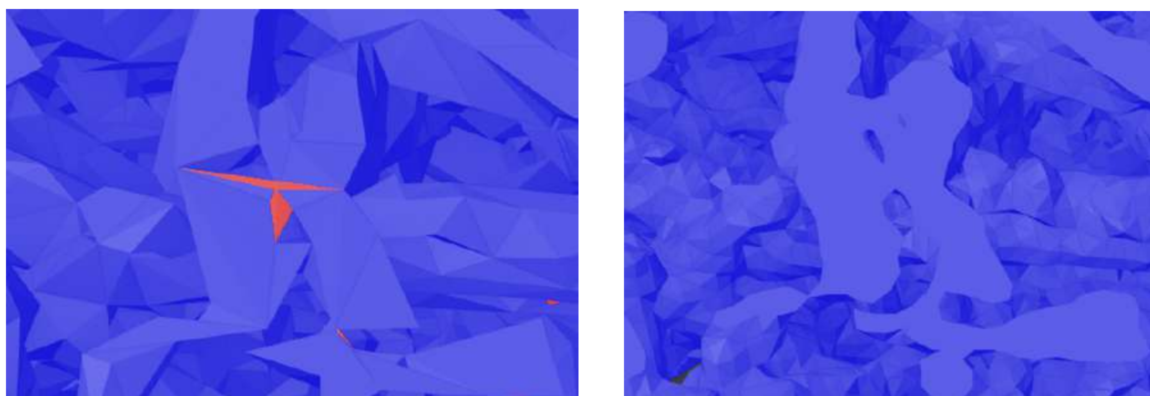
fig. 12 : exemples de décimation sur des fibres non imprégnées (dataset KPP Prop non imprégné)

Géométries dégénérées

La décimation peut entraîner des problèmes de cohérence géométrique, où les normales se trouvent inversées par endroit. Sur les figures 13 et 14, les couleurs bleues et rouges donnent l’orientation des normales des faces. On voit apparaître des problèmes (toutefois assez rares) aux niveaux de décimation les plus bas.

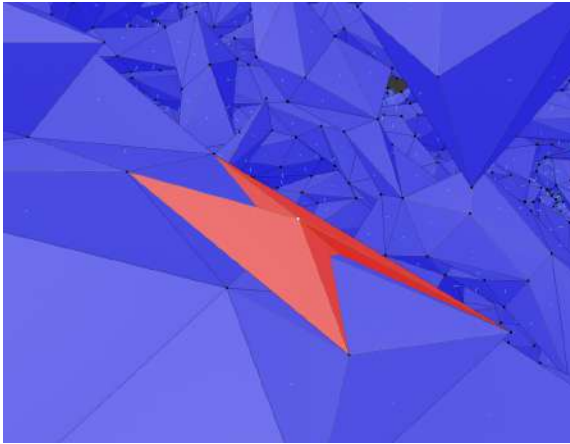
Blender dispose d’outils pour réparer ces erreurs, mais ils ne sont pas parfaits et passent parfois à côté de certains cas particuliers.

On répare aussi les faces et arrête avec une taille nulle dans cette étape.



Décimation 10 % Aucune décimation

Fig. 13


Exemple d’une face autointersectant à l'extérieur, après décimation

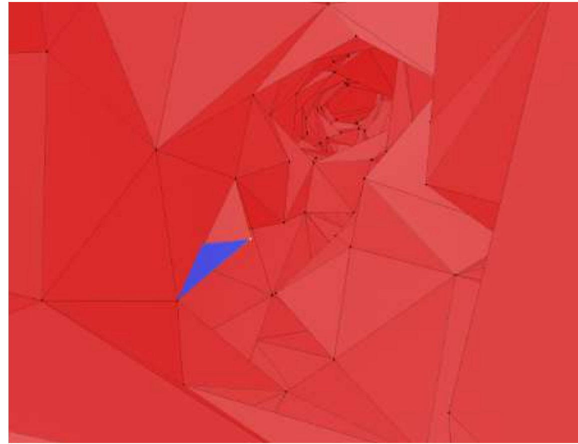

Exemple d’une face autointersectant à l’intérieur, après décimation

Fig. 14

Composantes connexes

La séparation des composantes non connexes est également un traitement qui s’avère utile et permet de détruire tous les petits morceaux insignifiants du mesh.

Prenons garde, on voit sur la figure 15 que la plus grande composante (en couleur) ne représente qu’une petite partie du mesh. Pour remédier à ce problème, une option pour garder les N plus grandes composantes a été ajoutée. On peut aussi utiliser un seuil au-delà duquel on considère que l’îlot est assez grand et qu’il faut aussi le garder.

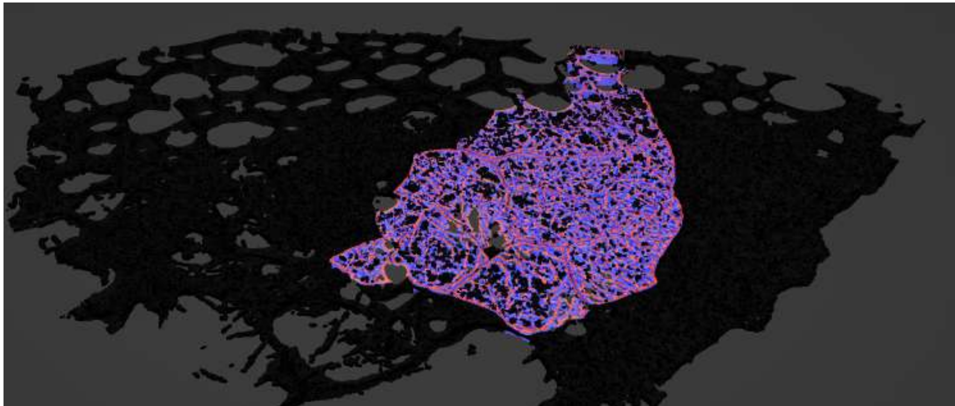


Fig. 15 : En couleur, la plus grande composante connexe sur 15 couches. En revanche, les grands îlots ont tendance à se connecter lorsqu’il y a un nombre plus important de slices.

Billes de cuivre

Le but étant de simuler les interactions avec les dépôts de cuivre, le dernier traitement implémenté consiste à distribuer des billes de cuivre sur le mesh.

Plutôt qu’un nombre prédéfini de billes, on utilise une densité de distribution sur tout le mesh.

Les billes sont distribuées sous Blender avec des *geometry nodes* avec des tailles aléatoires entre des valeurs min et max. Ensuite, un script déplace les centres selon la normale (pour que les billes reposent sur la surface) et exporte les données dans un fichier texte pour la partie simulation.

La figure 16 représente une distribution moyenne sur une éponge de menger.

Cette méthode de distribution était utile durant le projet, mais la création des billes a finalement été directement intégrée dans la simulation.

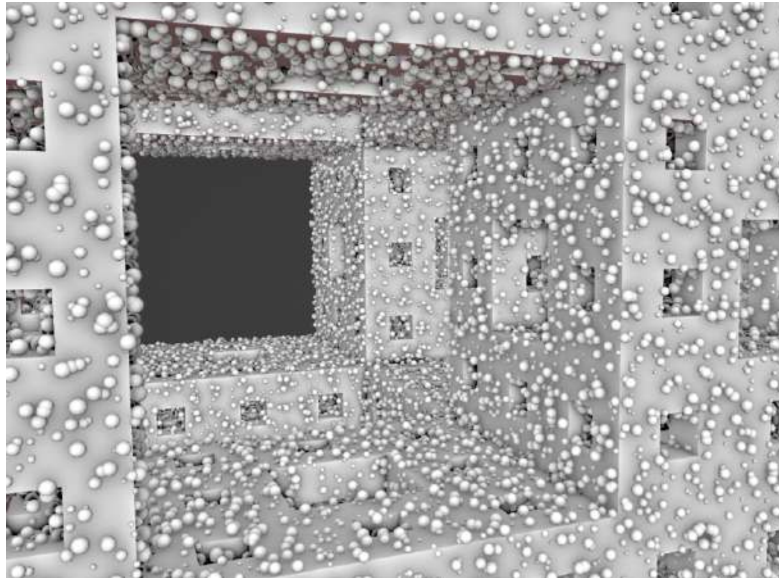


Fig. 16 : Visualisation des billes de cuivre

Autres traitements

Un autre traitement du meshes est la découpe dans un cube unitaire pour ne garder que la partie intéressante à la simulation. Ce traitement a été abandonné, car déjà présent sous une autre forme dans la partie simulation.

Limites

Actuellement la limite n'est plus sur la génération du mesh comme avant, mais sur le traitement. En effet, à partir de 200M triangles, Blender commence à crasher souvent à cause de l'empreinte mémoire (sur une machine 32 GB).

Il faut compter :

- **1 h** de traitement pour **200M**,
- **30 min** pour **140M**
- **30 s** pour **4M**

L'augmentation déraisonnable du temps de traitement est due à l'utilisation du swap mémoire, et peut être minimisé sur des machines avec une plus grande capacité.

Visualisation

Pour éviter d'avoir à attendre trop longtemps ou de charger en mémoire un modèle complet, un outil de visualisation a été créé. On ne se base que sur la séquence d'image et grâce à un shader qui charge ces images, on utilise Blender pour rendre un volume de voxels.

Les figures 17 à 19 montrent cette représentation 3D sur quelques sets de données :

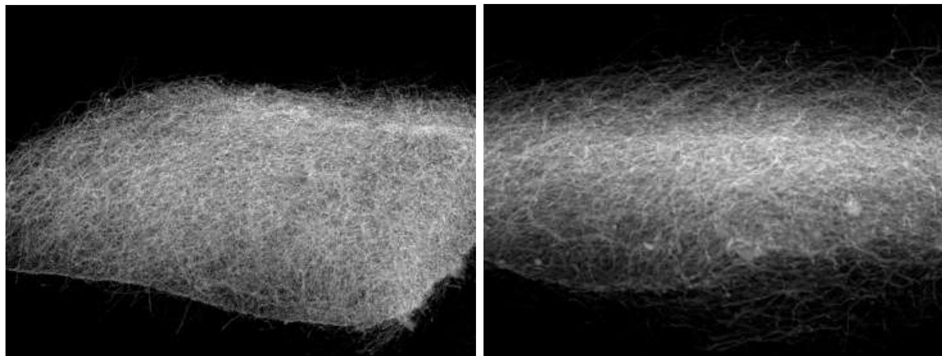


Fig. 17 : Vue des fibres non imprégnée (dataset KPP Prop non imprégné)

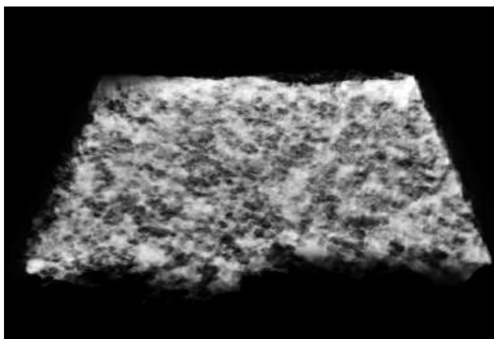


Fig. 18 : (dataset KPP Prop AHPCS dilue TT 1000)

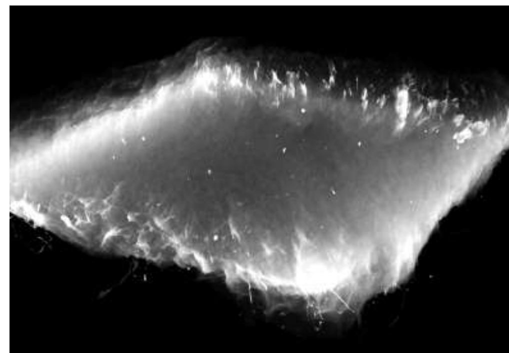


Fig. 19 : (dataset KPP Prop AHPCS)

Voici, par exemple, une Animation [5] calculée en quelques minutes grâce à cette visualisation. À noter que cela marche aussi avec les images en couleurs.

Cette technique est surtout utile pour prévisualiser un prétraitement des images sur un set de données sans avoir à générer et à traiter un mesh de plusieurs millions de triangles.

[5] https://drive.google.com/file/d/1M4uCHvkO00Q3ORrIwHFVU7F9awlJwCIi/view

Résultats

Nous disposons donc maintenant d'une suite d'outils qui permettent de partir d'une séquence d'image, et d'arriver à un mesh utilisable pour la simulation de fluide.

On utilise en fin de compte une approche uniquement surfacique, car le volumique n'est pas adapté à la taille des données que l'on gère.

Le prétraitement des images n'est finalement pas important pour obtenir des maillages corrects.

La limite actuelle est la phase de traitement du mesh (empreinte mémoire et temps de traitement).

On arrive à obtenir sans difficulté des maillages de plusieurs dizaines de millions de polygones comme illustré sur les figures 20 à 23.

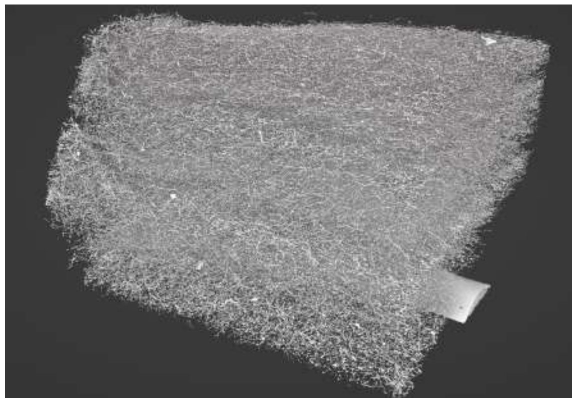


Fig. 20 : KPP Prop TEOS TT 1400

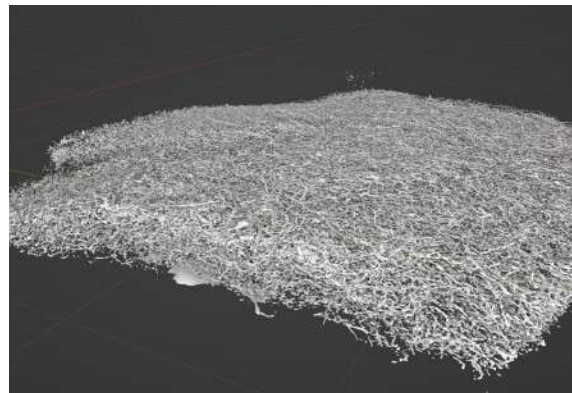


Fig. 21 : KPP Brut TEOS TT 1400

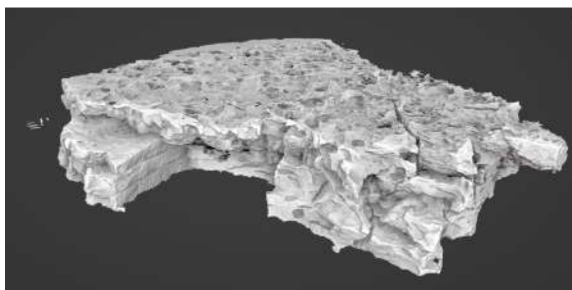


Fig. 22 : TT 1000 KPP Prop 7 couches

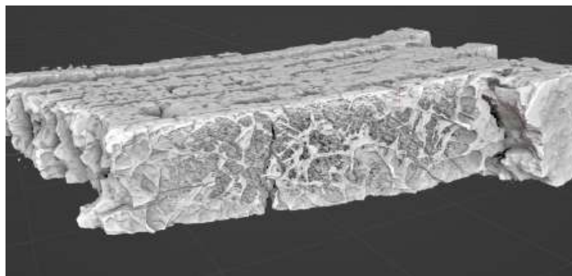


Fig. 23 : TT 1000 KPP Brut 7 couches

Perspectives

Puisqu'on n'utilise plus les informations volumiques, on passe à côté d'optimisations possibles dans la partie simulation. En particulier grâce aux relations d'adjacences entre les cellules qui divisent l'espace pour simuler les particules.

On pourrait utiliser un outil comme Tetgen [6] pour créer des tétraèdres dans le mesh, et ainsi avoir un découpage en zones volumiques. Cela accélérerait le calcul d'intersection avec la surface en faisant avancer le rayon le long des tétraèdres, et aussi, on n'aurait à tester que les particules du tétraèdre actuel et des tétraèdres voisins pour les collisions entre particules.

[6] https://wias-berlin.de/software/index.jsp?id=TetGen&lang=1

Guide d’utilisation

Génération d’un mesh

Pour générer un mesh, on lance le script `marching_cubes_tomographie.py` depuis la racine du projet. Au lancement, on peut spécifier des options comme le chemin vers la séquence d’images, le fichier de sortie, le nombre d’images à charger, l’image de départ, la réduction de la résolution, l’isovaleur, et le padding.

Exemple : `./src/marching_cubes_tomographie.py --ISO_LEVEL=64 --RES_MULT=0.33`

Le maillage sera exporté dans un fichier obj.

Traitement d’un mesh

Pour le traitement, des scripts automatisés sont disponibles. Il faut lancer le fichier `bpy_scripts.blend` avec **Blender 3.2+**.

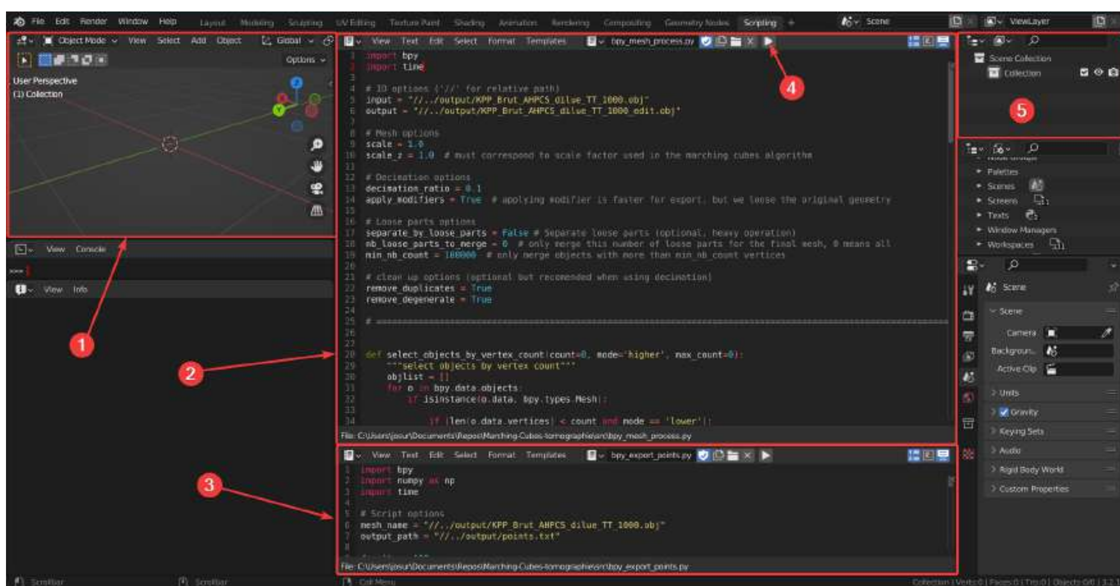


Fig. 24 : Interface de traitement dans Blender

Sur la figure 24, on a :

1. La vue 3D dans laquelle va apparaître les meshes que l'on importe et traite
2. Le script de traitement
3. Le script de génération des billes de cuivre
4. Le bouton pour lancer le traitement
5. La liste des objets de la scène pour nettoyer au besoin

Au début du script (2.), il y a des options de traitement que l’on peut modifier avant de le lancer.

L’importation et l’exportation se font automatiquement.

Si le script est édité en externe, il faut le recharger dans Blender à l’aide du bouton rouge ( ) qui apparaîtra en haut de la zone 2.

Pas besoin de sauvegarder le fichier après le traitement, on préfèrera éditer le fichier `bpy_mesh_process.py` plutôt

On peut lancer la console Blender pour voir l’avancement du script et les timings comme illustré sur la figure 25.

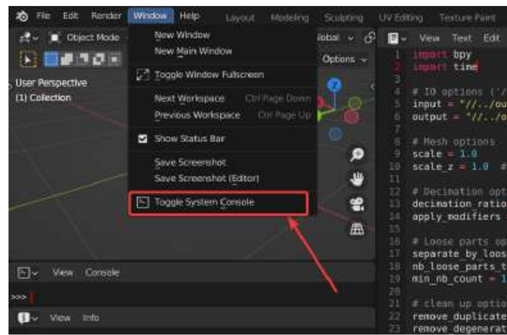


Fig. 25 : Comment lancer la console Blender

Visualisation

Pour lancer une visualisation, on utilise le fichier `oslVoxel.blend`.

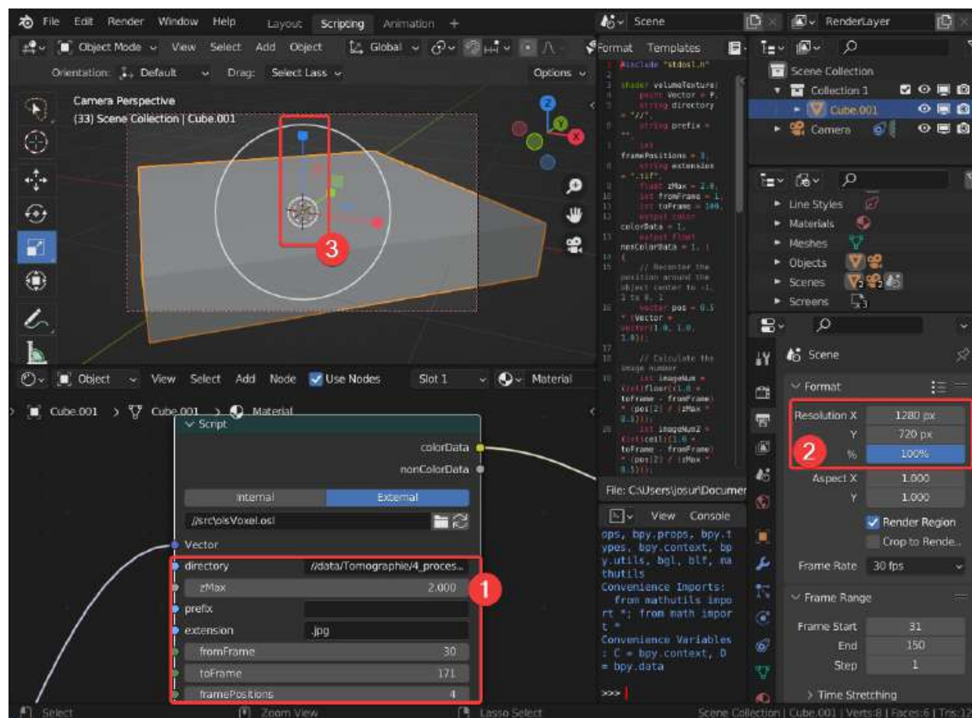


Fig. 26 : Interface de `OslVoxel.blend`

Sur la figure 26, on a :

1. Paramètre de chargement de la visualisation
2. Changement de la résolution
3. Guizmo pour changer la taille verticale du cube de visualisation (en bleu)

Dans l'encadré (1.) on spécifie le chemin (relatif avec "`//`") vers la séquence d'image, le préfixe de chaque image (par exemple : "`Slice`") et l'extension. Les images de début et de fin. La dernière option représente le nombre de chiffres après le préfixe qui contiennent l'information sur le numéro de l'image.

Pour calculer une image, on utilise **F12** (**ATL+S** pour enregistrer l'image), et pour une animation **CTRL+F12** (sortie en `mp4`).

Simulation

Pour notre projet, il était logique de proposer une simulation afin de fournir une visualisation des phénomènes attendus. Pour réaliser cette simulation, nous ne sommes pas partis de zéro, mais d'une autre simulation de particules proposée par NVIDIA[7]. Cette simulation nous permettait de ne pas coder l'architecture du projet ainsi que les interactions des particules de fluides entre elles. Mais il faut donc ajouter plusieurs éléments afin que cette simulation nous satisfasse. Nous allons de ce fait vous décrire les principaux ajouts par rapport à la simulation de base.

Importation d'objets 3D :

Une des étapes importantes de la simulation était de pouvoir importer des objets 3D (modélisés par Josué). Grâce à ça, on va pouvoir visualiser et simuler sur des objets appropriés. L'importation se fera à l'aide de fichier .OBJ et l'importer d'Assimp[8]. Au lancement du programme, assimp va ainsi lire le fichier .OBJ et va pouvoir remplir un objet TriangleMeshModel. La classe TriangleMeshModel, va nous permettre de regrouper tous les éléments de l'objet 3D. Il faut savoir qu'un TriangleMeshModel contient potentiellement plusieurs TriangleMesh. En revanche, pour l'instant, un seul est utile, car on a un objet unique et connexe. Cette classe contenant toutes les informations nécessaires va donc nous servir pour deux choses différentes :

- L'affichage de l'objet grâce à OpenGL [9] , les normales donnent la couleur comme sur la figure 27.
- L'interaction Particules/objet

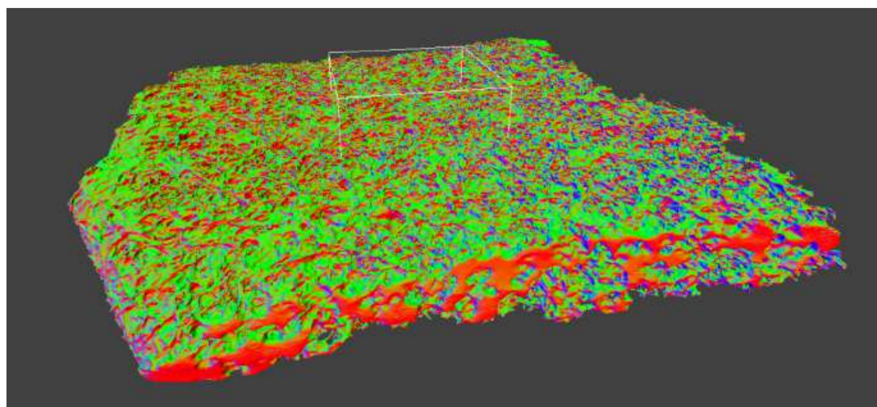


Fig. 27 : Affichage de l'objet (4 Millions de triangles) grâce à OpenGL

Interaction particules/objets

Une fois l'objet 3D affiché, la prochaine étape, c'est de l'intégrer correctement à la simulation, c'est-à-dire vérifier que les particules de la simulation interagissent avec lui et qu'elles ne passent plus passer au travers.

Pour ce faire, nous avons rempli des buffers à l'aide de notre variable objet initialisée précédemment afin de les envoyer au GPU. Il faut savoir que l'on va avoir besoin de GPU dans le but de calculer toutes ces interactions. Alors, on va séparer le travail sur plusieurs threads, un à un les threads vont avoir une particule courante. Il va ainsi tester plusieurs éléments, par exemple, s'il touche une autre particule, s'il touche un bord et lui affecter le facteur de gravité par exemple. Il faut donc rajouter à la

[7] https://developer.download.nvidia.com/assets/cuda/files/particles.pdf

[8] https://github.com/assimp/assimp

[9] https://www.opengl.org/

fin de ces traitements notre calcul d'interaction avec les triangles, et ainsi modifier la vélocité de la particule courante.

On va alors devoir faire un test d'intersection particules/triangle. Pour savoir s'il y a une intersection entre une particule et un triangle, nous allons tirer un rayon à partir du centre de la sphère en direction de sa vélocité. On va ensuite utiliser des solutions géométriques[10] pour savoir si le rayon intersecte le plan du triangle et enfin s'il intersecte le triangle. Ce calcul va être divisé en deux grandes étapes. La première va consister à trouver un point P qui va être l'intersection du rayon sur le plan du triangle. Dans un premier temps, on va éviter deux cas simples, le premier le cas où le rayon est parallèle au plan et l'autre celui ou le rayon est derrière le triangle. Une fois ces deux cas éviter, on va pouvoir calculer T qui est la distance entre l'origine du rayon et le point d'intersection, ainsi en injectant T dans l'équation du rayon, on peut obtenir le point P . À partir de ce moment-là, on va avoir un point P qui est sur le plan du triangle, ainsi, il va falloir savoir s'il est à l'intérieur du triangle ou à l'extérieur. Pour ce faire, on va regarder si pour chaque arrête le point est du bon côté. S'il est du bon côté pour les trois arêtes, alors le point est à l'intérieur du triangle.

Ensuite, si on détecte une collision, on va pouvoir corriger la vélocité de la particule afin de la replacer et qu'elle ne passe pas à travers du triangle. Pour corriger cette vélocité, on va utiliser la réflexion pour avoir la direction du rebond, pour ce faire, nous allons utiliser le vecteur de direction incident et la normale du triangle (figure 28).

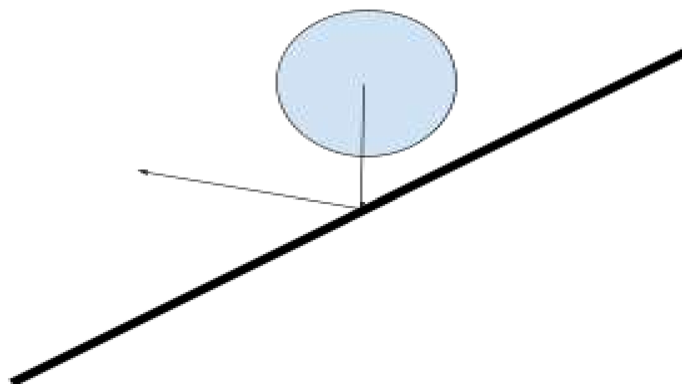


Fig. 28 : Exemple de rebond de la particule sur un plan

Optimisations

Pour la simulation, nous utilisons une quantité énorme de données, par exemple, des objets à 4 millions de triangles et une simulation à 1 million de particules. Si nous ne faisions aucune optimisation pour l'interaction triangle/particules, alors il faudrait que chaque particule teste les 4 millions de triangles à chaque itération. Cela va faire une quantité gigantesque de tests d'interaction si l'on procède sans optimisation. On doit donc en premier temps trouver des moyens pour optimiser le temps de calcul de chaque itération.

---

10 https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution

Optimisation du temps

Nous sommes partis sur l'idée d'une grille uniforme afin d'optimiser l'interaction triangle particule. Celle-ci va diviser l'espace de la simulation en une grille composée de cellules cubiques 3D de même taille. Par exemple, si l'on prend une grille de taille 128 × 128 × 128, alors, on aura 2 097 152 cellules. Vous pouvez voir un exemple de cette division figure 29.

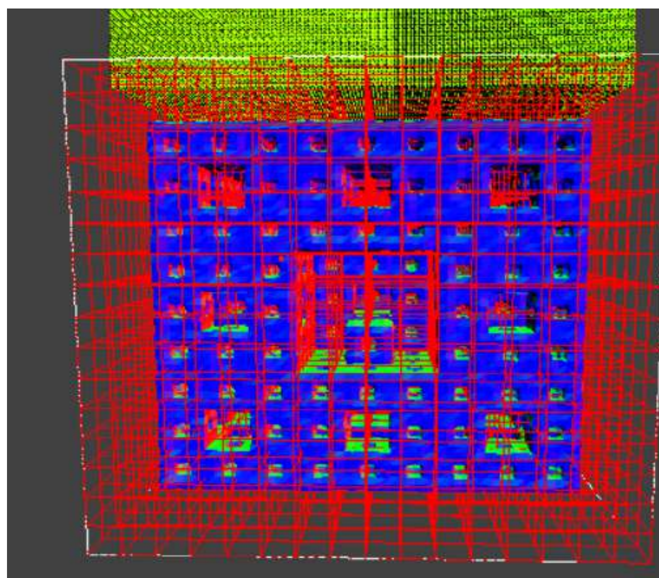


Fig. 29 : Affichage de l’espace divisé en cellule

Grâce à cette technique, pour une particule courante, on va pouvoir lui demander de tester l’interaction seulement avec les triangles présents dans sa cellule et les cellules voisines. On aura ainsi un total de 27 cellules à tester sur plus de 2 millions. On va de ce fait éviter beaucoup de tests inutiles, mais il va falloir savoir, pour un point de l’espace donné, savoir dans quelle cellule on se trouve.

On va ainsi avoir une fonction qui va nous retourner une position XYZ de cellule dans la grille, et une autre qui va convertir les coordonnées XYZ en un seul int. Grâce à cette valeur, on va pouvoir référencer un endroit spécifique d’un tableau pour situer les indices des triangles présents dans la cellule courante.

Nous avions d’autres idées d’optimisation que nous avons tenté d’implémenter, mais qui prennent plus de temps à développer. Par exemple, au lieu de diviser l’espace, diviser seulement AABB [11] (voir figure 30 et 31) de l’objet. Cela pourrait permettre d’effectuer la simulation, pas uniquement dans un cube de taille 2, mais sur une plus grande partie de l’objet.

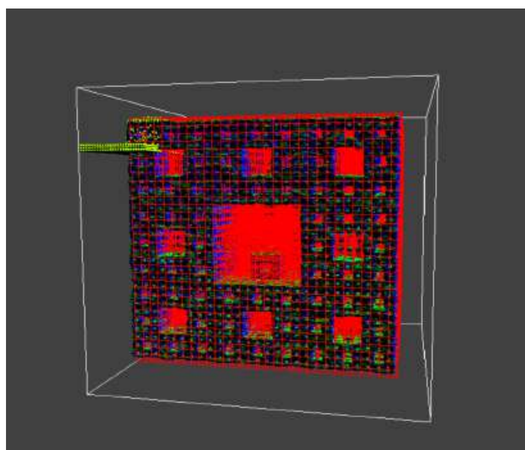


Fig. 30 : Diviser l’AABB du cube de menger

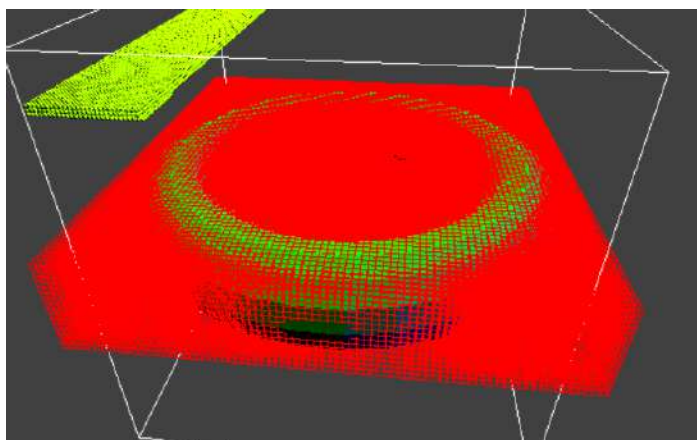


Fig. 31 : Diviser AABB d’un objet non cubique

[11] Boîte englobante de l’objet

Optimisation de mémoire

Grâce à cette optimisation, le temps de calcul pour chaque frame est considérablement amélioré. Il reste un problème, cela prend énormément de place en mémoire. En effet, pour cette solution, nous utilisons un moyen très coûteux en mémoire. Pour cette méthode, chaque cellule va avoir un nombre fixe d'emplacements de triangle libre. Par exemple, on aura une cellule qui peut contenir au maximum 500 triangles. De ce fait, la première cellule aura de la case 0 à 499 libres, la seconde 500 à 999, et ainsi de suite (figure 32).

Ceci est un problème pour deux raisons :

- Si l'on est sur une cellule ne contenant pas ou contenant peu de triangle, alors on va prendre de la place mémoire pour rien.
- Si l'on est sur une cellule qui contient beaucoup de triangle, ainsi, on peut dépasser la variable, et ainsi ne pas pouvoir référencer tous les triangles.

C1					C2					C3					
5	8	4	-1	-1	0	2	7	4	1	-1	-1	-1	-1	-1	...

Fig. 32 : Exemple de la grille sans optimisation

Si l'on prend pour exemple le tableau ci-dessus ou le nombre max de triangle par cellule est 5 et où C1,C2 et C3 représentent respectivement la cellule 1,2 et 3. Pour la Cellule 1 on aura donc les triangles d'indice 5,8 et 4 de présent. Cette cellule contient deux cases vides représentées par les -1. Pour la cellule deux, on a le second problème représenté. En effet, on a rempli toutes les cases disponibles par cellule, il se peut que des triangles ne soient pas renseignés. Pour la cellule 3, il n'y a aucun triangle dans la cellule, mais on prend quand même la place en mémoire…

C'est pour cela qu'on va avoir besoin d'optimiser la mémoire. Nous avons ainsi eu une idée utilisant deux tableaux. Notre premier tableau va être de taille 2* le nombre de cellules. Pour chaque cellule, il va nous renseigner l'indice de début à lire pour les triangles de cette cellule et le nombre de valeurs qu'on doit lire. On va pouvoir ainsi se reporter dans le second tableau et commencer à lire au bon endroit le nombre de valeurs associées (figure 33).

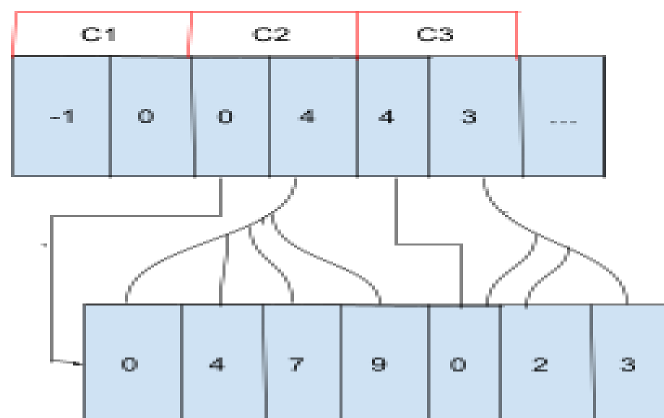


Fig. 33 : Exemple d'optimisation de la grille

Voici l'exemple de cette optimisation dans le tableau ci-dessus. Dans notre premier tableau, on va avoir deux cases par cellules de la grille. La première valeur va être l'indice de début du second tableau, et la seconde le nombre de valeurs à lire. Pour la cellule 1, on va donc avec un -1 en indice de début, cela veut dire qu'il n'y a aucun triangle dans la cellule. Pour la cellule deux, on va avoir comme indice de début zéro et comme nombre de valeurs à lire 4. On va ensuite commencer dans le second tableau à l'indice zéro et lire quatre valeurs. Et c'est le même raisonnement pour la cellule 3. Grâce à cette technique, on constate ainsi que tous les problèmes sont résolus.

Les billes de cuivre

Une fois que la simulation tourne, il faut savoir ce que l'on veut montrer. Il faut savoir que pour le côté matériau, ce qui est intéressant, c'est que le fluide va interagir avec des billes de cuivre qui vont rester collées à la paroi de la céramique. On va ainsi essayer de modéliser cela pour un résultat final.

Au départ, nous étions partis sur une solution qui utiliserait un fichier texte en plus à importer en même temps que l'objet. Celui-ci contiendrait les positions et rayons relatifs aux billes de cuivre. Mais cette solution commençait à poser un problème sur de gros objets 3D. Nous avons donc décidé de modéliser ceci directement lors de la situation (figure 34).

Dorénavant, quand nous allons lancer la simulation, le programme va alors choisir aléatoirement des triangles présents dans l'espace de simulation. Ce sont sur ces triangles qu'on va "simuler" les billes de cuivre avec des rayons différents. On va ainsi pouvoir visualiser ces billes grâce à OpenGL qui au départ vont être blanches, mais quand nous arrêterons la simulation prendront deux teintes différentes en fonction de leur nombre d'interactions.

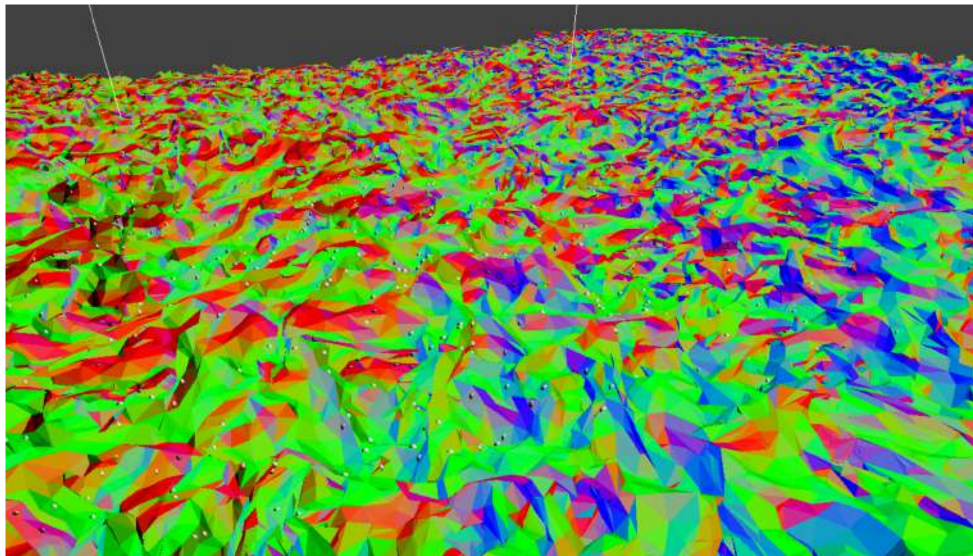


Fig. 34 : Modélisation des billes de cuivre au milieu des triangles sélectionner

Pour optimiser la partie calcul, nous sommes partis sur la même solution, c'est-à-dire l'utilisation de la grille.

Guide d'utilisation

Quand vous lancez le programme, cela va d'abord importer l'objet 3D, il faut en conséquence attendre un certain temps en fonction de l'objet. Ensuite, vous arriverez sur la scène globale. Vous pouvez voir un cube de taille 2 centré en 0,0,0. Celui-ci va être l'espace de simulation, autrement dit là où les particules vont se mouvoir. Vous verrez aussi l'objet 3D que vous avez importé.

Les étapes de la simulation

- **1re étape :** Comme dit précédemment, l'espace de simulation n'est que dans le cube, il va falloir donc déplacer votre objet en fonction que la zone qui vous intéresse soit dans ce cube. Pour ce faire, il y a des sliders qui vont vous permettre de le déplacer en X, Y et Z.
- **2e étape :** Une fois votre objet au bon endroit, il vous suffira d'appuyer sur "L". Cela peut prendre un certain temps en fonction des triangles pris en compte pour la simulation. Durant ce temps, le programme va charger les buffers pour les triangles et les billes de cuivre pour tout envoyer au GPU pour le traitement de la simulation.
- **3e étape :** Après ce court instant, la simulation va se lancer, et les particules vont traverser l'objet de haut en bas. À Savoir qu'il est possible d'enlever l'affichage de l'objet afin de voir le peuplement de celui-ci en appuyant sur "D" (figure 35).
- **4e étape :** Quand vous voulez lors de la simulation, vous pouvez appuyer sur "R" pour réinitialiser. C'est-à-dire que les particules vont reprendre leurs positions initiales, et vous pourrez de nouveau déplacer l'objet. Vous constaterez que les billes de cuivre seront colorées de deux couleurs différentes. Celles en bleu seront celles où les particules n'auront pas assez interagi avec en fonction de la moyenne de toutes les interactions et en rouge, c'est quand le nombre d'interactions dépasse la moyenne.(figure 36 et 37)
- **5e étape :** Et ainsi de suite, on peut retourner à l'étape 2

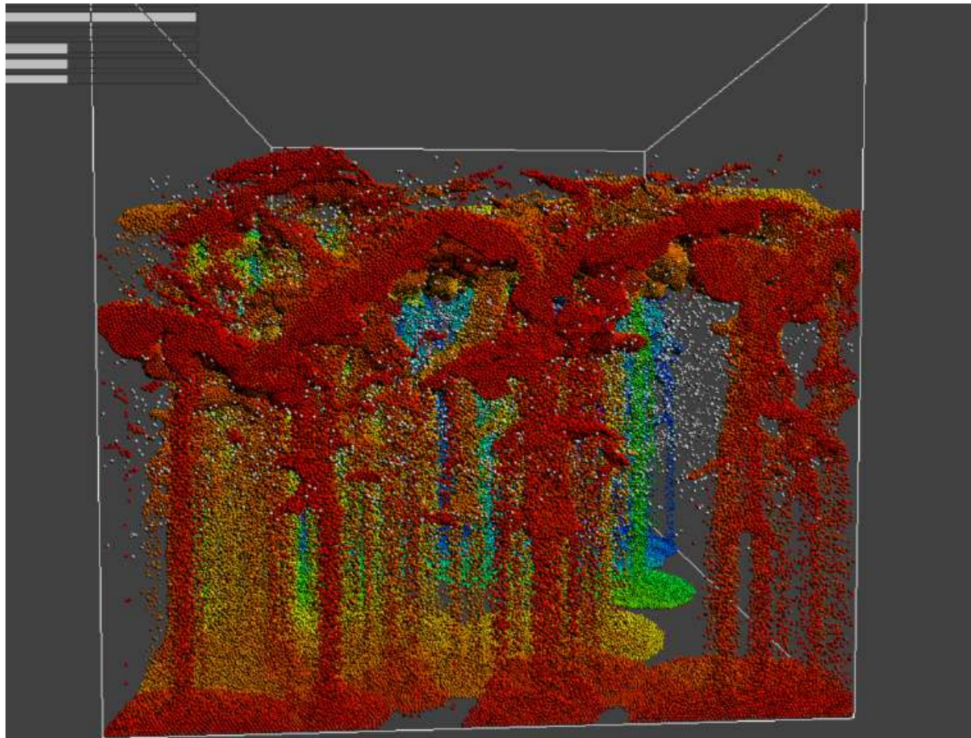


Fig. 35 : Simulation en désactivant l'affichage de l'objet

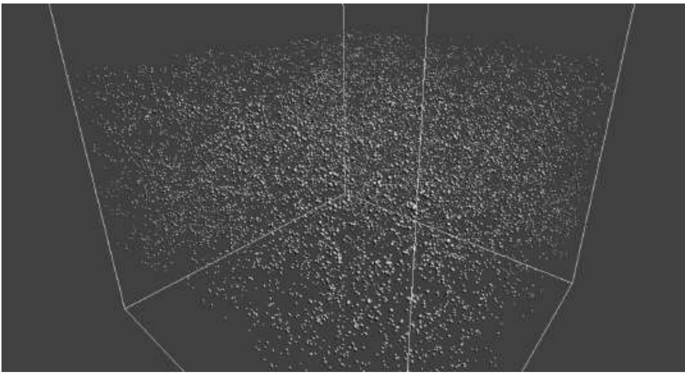


Fig. 36 : Les billes de cuivre avant les résultats

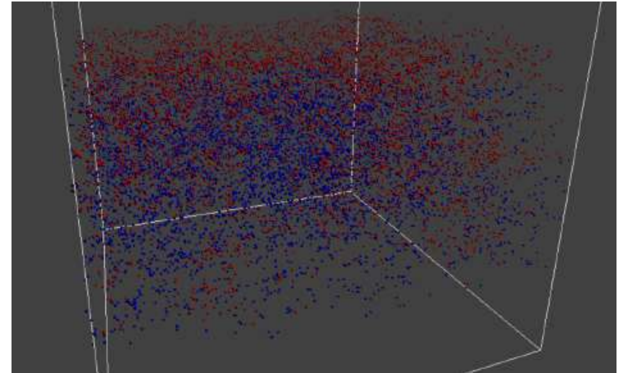


Fig. 37 : Billes de cuivre avec résultat

Les choses à savoir

Il faut savoir qu'il est possible de changer plusieurs paramètres durant la simulation, vous pouvez voir les sliders disponible figure 38 :

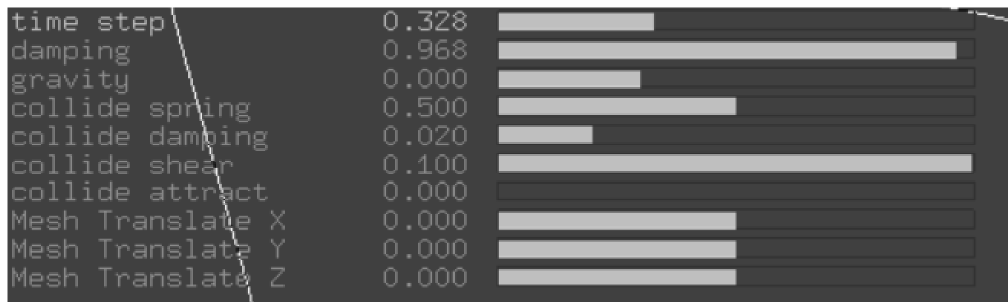


Fig. 38 : Les sliders disponibles

Les variables collide vont être utilisées si vous voulez représenter différents types d'interactions pour les collisions particules/particules. Les mesh translate vont être pour déplacer l'objet lors de l'initialisation.

Pour la version CUDA, nous utilisions la 11.4. Si vous voulez récupérer le projet, il faut alors le clone depuis ce dépôt git.

Le damping va nous permettre de simuler la réduction d'amplitude. C'est-à-dire, par exemple, une boule qui va rebondir sur un plan, au fur à mesure du temps, ne va plus rebondir grâce aux frottements par exemple. Le time step est la durée d'avancement de la simulation, par exemple, pour ralentir ou accélérer les particules, le temps va s'écouler plus vite si on l'augmente.

Sur cette simulation, il est aussi possible de déplacer la caméra afin de voir les phénomènes que l'on veut. En laissant appuyé sur le clic gauche et déplaçant la souris, on peut effectuer une rotation autour du point observé. À l'aide du clic molette, on va pouvoir déplacer la caméra. En laissant appuyer sur CTRL et le clic gauche, on va dézoomer ou zoomer à l'aide du mouvement de la souris.

Il faut savoir que le projet de base utilise une vieille version d'OpenGL. Cette version est utilisée pour l'affichage des particules et des billes de cuivre.

Vous pouvez changer l'objet 3D importé lors de la simulation en renseignant le nom de celui-ci dans la variable "MESH_NAME" qui se situe en haut du fichier particles.cpp. Il faut aussi que cet objet soit présent dans le dossier "data" du projet. Au même endroit que la variable "MESH_NAME" vous pourrez modifier divers paramètres comme le nombre de cellules utilisées pour la grille, le nombre de particules…

Démonstration

Dans la figure 39 ci-dessous, vous pouvez voir, une vidéo complète de la simulation :

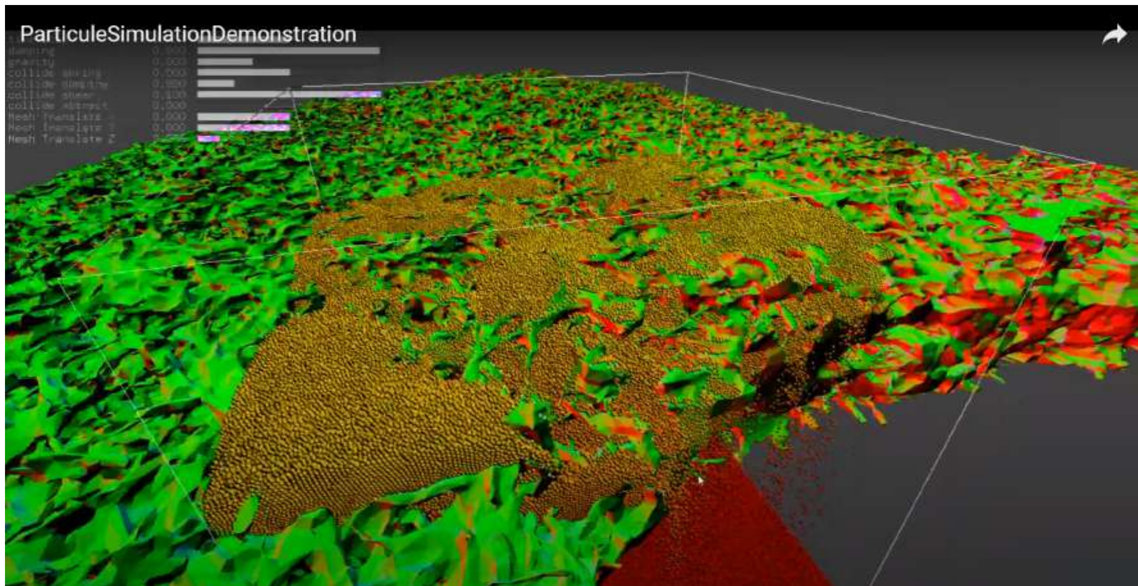


Fig. 39 : Démonstration de la simulation
(https://drive.google.com/file/d/13x8C62ZhZT8t5KvsJfdz2WhhMRMZYmsc/view)