Chef d'Oeuvre

Traitement de la géométrie : décomposition de maillages en variétés topologiques

Recette

Laura BARROSO
Martin BOUYRIE
Sébastien EGNER

Encadrant : Nicolas MELLADO
Clients : Nicolas MELLADO, Loïc BARTHE

Université Toulouse III - Paul Sabatier
28 février 2021

1 Introduction

Le problème de conservation d'une surface **manifold** s'est manifesté lorsque le besoin de réaliser un traitement sur une surface **non manifold** est apparu. Une surface **non manifold** peut survenir à la suite d'un traitement par un algorithme ou volontairement par un artiste par exemple.

Bien que le problème survienne depuis maintenant plus d'une vingtaine d'années, il n'existe pas encore de méthode de correction de maillages sans contraintes. Cependant, puisque c'est un problème conséquent, de nombreuses recherches sont effectuées pour parvenir à un résultat satisfaisant. L'article de ATTENE et al. [2013] présente notamment différentes techniques et méthodes utilisées. Il compare ainsi leur différences et ce qu'elles apportent pour une meilleure utilisation de ces méthodes.

Nous présentons dans ce papier, le résultat de notre application des méthodes de GUEZIEC et al. [2001] permettant la correction d'une surface **manifold** en une surface **non manifold**. Deux approches sont présentées dans ce papier, une approche de destruction et une approche de reconstruction. La première approche est appellée **cutting** et la seconde **stitching**. L'approche par cutting permet de corriger une surface **non manifold** en une surface **manifold** par duplication de sommets, on agit alors à l'échelle topologique de la surface. L'approche par stitching quant à elle, est la continuité de l'approche par cutting, mais propose une reconstruction supplémentaire de la topologie du maillage afin que la surface soit **manifold** là où le cutting ne fait que dupliquer. Chaque approche a deux méthodes différentes permettant ainsi une efficacité plus significative selon la surface rencontrée. Pour le cutting, ces deux méthodes sont le **cutting local** et **le cutting global** et pour le stitching, ces deux méthodes sont le **pinching** et le **snapping**.

2 Manuel d'utilisation

Le projet est disponible sur notre dépôt Gitlab.

2.1 Prérequis

Notre application requiert l'installation de plusieurs logiciels et bibliothèques afin de fonctionner correctement. Voici les différents prérequis :

- Un compilateur C++ supportant le standard C++17 (MSVC sous Windows et gcc sous Linux par exemple)
- CMake (version 3.6 minimum)
- Git
- Ninja (sous Windows uniquement)

Il va également falloir installer Radium-Engine, installation que nous allons détailler car il existe quelques spécificités à connaître sur l'installation de Radium-Engine pour notre projet. En effet, la classe *TopologicalMesh* de Radium-Engine n'ayant pas été pensée pour fonctionner avec des maillages **non**

manifold, nous avons dû modifier certains fichiers et travailler sur un *commit* spécifique.

2.2 Installation de Radium-Engine sous Windows

Nous avons eu quelques soucis au départ pour installer Radium-Engine car les dépendances ne compilaient pas correctement lorsqu'on laissait Radium-Engine s'occuper de leur installation. Une *issue* Github a été remontée à ce sujet et nous avons procédé de manière différente pour l'installation des dépendances (nous nous sommes inspirés de ce qui était fait par l'outil d'intégration continue de Radium-Engine). Voilà donc comment installer les dépendances de Radium-Engine, sous Windows :

Listing 1 – Installation de Radium-Engine sous Windows

```
cd BASE_DIRECTORY
git clone https://github.com/STORM-IRIT/Radium-Engine
git clone https://gitlab.com/Storm-IRIT/radium-private
    -applications/radium-mesh-repair.git
cd Radium-Engine
git checkout d03c4881c97cfba88084f2cac10d17d56b7a1850
cd ..
move radium-mesh-repair/external/TopologicalMesh.cpp
    Radium-Engine/src/Core/Geometry/TopologicalMesh.cpp
mkdir Radium-Engine-external
cd Radium-Engine-external
mkdir build
mkdir install
cd build
cmake ..\..\Radium-Engine\external -GNinja -
    DCMAKE_CXX_COMPILER=cl.exe -DCMAKE_C_COMPILER=cl.
    exe -DCMAKE_BUILD_TYPE=Release -
    DCMAKE_INSTALL_PREFIX=../install/
cmake --build . --config Release
cd ..\..\Radium-Engine
mkdir build
cd build
cmake .. -GNinja -DCMAKE_CXX_COMPILER=cl.exe -
    DCMAKE_C_COMPILER=cl.exe -DCMAKE_BUILD_TYPE=Release
     -DQt5_DIR=QT5_INSTALLATION_DIRECTORY\msvc2019_64\
    lib\cmake\Qt5 -DEigen3_DIR=BASE_DIRECTORY/Radium-
    Engine-external/install/share/eigen3/cmake/ -
    DOpenMesh_DIR=BASE_DIRECTORY/Radium-Engine-external
    /install/share/OpenMesh/cmake/ -Dglm_DIR=
    BASE_DIRECTORY/Radium-Engine-external/install/lib/
    cmake/glm/ -Dglbinding_DIR=BASE_DIRECTORY/Radium-
    Engine-external/install/share/glbinding/ -
```

```
    Dglobjects_DIR=BASE_DIRECTORY/Radium-Engine-
    external/install/share/globjects/ -Dstb_DIR=
    BASE_DIRECTORY/Radium-Engine-external/install/
    include/stb/ -Dassimp_DIR=BASE_DIRECTORY/Radium-
    Engine-external/install/lib/cmake/assimp-5.0/ -
    Dtinyply_DIR=BASE_DIRECTORY/Radium-Engine-external/
    install/lib/cmake/tinyply/ -Dcpplocate_DIR=
    BASE_DIRECTORY/Radium-Engine-external/install/share
    /cpplocate -DCMAKE_INSTALL_PREFIX=../install/
cmake --build . --config Release --target install
```

Remarque : Vous pouvez préciser le compilateur de votre choix, ici **cl.exe** correspond au compilateur de Visual Studio. Dans le cas où ce dernier n'est pas reconnu, il vous faudra peut-être exécuter un script depuis la console pour configurer l'environnement, afin que *CMake* le reconnaisse ensuite. Ce script se trouve dans le répertoire d'installation de Visual Studio (par exemple, ici : "C :/Program Files (x86)/Microsoft Visual Studio/2019/Community/VC/Auxiliary/Build/vcvars64.bat").

2.3 Installation de Radium-Engine sous Linux

Sous Linux, l'installation de Radium-Engine est beaucoup plus simple et nécessite l'exécution des commandes suivantes :

Listing 2 – Installation de Radium-Engine sous Linux

```
cd BASE_DIRECTORY
git clone https://github.com/STORM-IRIT/Radium-Engine
git clone https://gitlab.com/Storm-IRIT/radium-private
    -applications/radium-mesh-repair.git
cd Radium-Engine
git checkout d03c4881c97cfba88084f2cac10d17d56b7a1850
cd ..
mv radium-mesh-repair/external/TopologicalMesh.cpp
    Radium-Engine/src/Core/Geometry/TopologicalMesh.cpp
cd Radium-Engine
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release -
    DCMAKE_INSTALL_PREFIX=../install/
make
make install
```

Remarque : Si vous obtenez une erreur de compilation avec Radium-Engine (une erreur liée à un attribut *position* inexistant dans le fichier *Radium-Engine/src/Gui/Timeline/TimelineScrollArea.cpp*), c'est que vous avez une version de Qt non compatible avec Radium-Engine. Cependant, cela peut facilement se résoudre en modifiant la ligne 157 de ce dernier fichier en remplaçant

position() par **pos()**.

2.4 Compilation de notre application sous Windows

Une fois Radium-Engine installé, on peut correctement installer notre application, de la manière suivante :

Listing 3 – Compilation de notre application sous Windows

```
cd BASE_DIRECTORY/radium-mesh-repair
mkdir build
cd build
cmake .. -GNinja -DCMAKE_CXX_COMPILER=cl.exe -
    DCMAKE_C_COMPILER=cl.exe -DCMAKE_BUILD_TYPE=Release
     -DQt5_DIR=QT5_INSTALLATION_DIRECTORY\msvc2019_64\
    lib\cmake\Qt5 -DRadium_DIR=BASE_DIRECTORY\Radium-
    Engine\install\lib\cmake\Radium -
    DCMAKE_INSTALL_PREFIX=.. ..
cmake --build . --config Release --target install
```

Une fois la compilation effectuée, l'application se trouve à l'emplacement **BASE_DIRECTORY/radium-mesh-repair/bin/MeshRepair.exe**.

2.5 Compilation de notre application sous Linux

Une fois Radium-Engine installé, on peut correctement installer notre application, de la manière suivante :

Listing 4 – Compilation de notre application sous Linux

```
cd BASE_DIRECTORY/radium-mesh-repair
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release -DRadium_DIR=
    BASE_DIRECTORY/Radium-Engine/install/lib/cmake/
    Radium -DCMAKE_INSTALL_PREFIX=.. ..
make install
chmod +x ../bin/MeshRepair
```

Une fois la compilation effectuée, l'application se trouve à l'emplacement **BASE_DIRECTORY/radium-mesh-repair/bin/MeshRepair**.

2.6 Exécution de notre application

Notre programme fonctionne en ligne de commande. Il prend deux paramètres obligatoires, qui correspondent au maillage d'entrée (sous format OBJ) et au fichier de sortie (lui aussi sous format OBJ). Le maillage d'entrée, censé être **non manifold** sera, après correction des singularités topologiques, enregistré à l'emplacement précisé par le maillage de sortie.

Voici par exemple un exemple d'utilisation de notre application :

Listing 5 – Exemple d'utilisation de notre application

```
./MeshRepair ../data/non_manifold/2_rectangles.obj ./
    output.obj
```

3 Description de la conception

Étant donné que nous avions entendu dire que notre projet serait possiblement intégré à Radium-Engine, nous avons fait attention à ce que notre code soit le plus modulable et intégrable possible. En accord avec notre tuteur de projet, nous avons utilisé une classe *MeshRepair* qui va s'occuper de la réparation des maillages. Celle-ci possède une interface extrêmement simple, uniquement composée d'un constructeur, qui prend en paramètre le type de *cutting* et le type de *stitching* qui sera utilisé pour la réparation. Elle possède également les trois fonctions **initialize**, **process** (utilisée pour gérer les faces **non manifold**) et **postProcess**, qui sont nécessaires si l'on souhaite réparer un maillage **non manifold**. Bien que ces fonctions soient publiques, elles ne sont pas destinées à être utilisées par l'utilisateur directement. Elles sont appelées en interne par Radium-Engine lorsqu'il tente de convertir un maillage **non manifold** en une instance de *TopologicalMesh*. La classe *MeshRepair* possède de nombreuses autres méthodes utilisées en interne, dont les fonctions sont les suivantes.

3.0.1 Construction et initialisation

- **MeshRepair** : Le constructeur de la classe. Prend deux paramètres optionnels, correspondant au type de *cutting* et au type de *stitching* qui seront utilisés. Ces deux paramètres sont respectivement du type **CUTTING_TYPE** et **STITCHING_TYPE**, qui sont tous les deux des énumérations qui simplifient l'interface et permettent d'éviter des erreurs de la part de l'utilisateur.
- **initialize** : La fonction appelée après que le maillage d'entrée ait été chargé (via une instance de *TriangleMesh*. Dans notre application, le corps de cette fonction est vide. Il est cependant nécessaire d'implanter cette fonction car elle sera appelée en interne par Radium-Engine.
- **process** : Cette fonction est appelée à la construction d'un *TopologicalMesh* à partir de notre maillage d'entrée, lorsque l'ajout d'une face entraîne une singularité topologique. Dans ce cas, cette face n'est pas ajoutée mais ses sommets le sont. Cette méthode est donc appelée à chaque fois que l'ajout d'une face est *impossible* et nous donne les *VertexHandle* de la face non ajoutée. En utilisant cette fonction, on peut donc garder une trace de ces faces qui n'ont pas été ajoutées et les traiter par la suite lors de la correction.
- **postProcess** : Cette fonction est appelée après la construction du *TopologicalMesh*, une fois que toutes les faces du maillage d'origine ont été ajoutées ou rejetées (via la fonction **process**). C'est dans cette fonction

que la réparation s'opère, en débutant tout d'abord par la préparation du maillage (via la fonction **prepare**), puis l'application du *cutting* et enfin, du *stitching*. Une fois cette étape terminée, le maillage de sortie est sauvegardé selon l'emplacement choisi par l'utilisateur.

— **prepare** : Cette fonction est appelée avant le prétraitement de notre maillage ; l'identification des singularités topologiques, et ajoute les différentes propriétés qui seront utilisées par la suite pour l'identification des différentes singularités topologiques de notre maillage (les **arêtes singulières** et les **sommets singuliers isolés**).

3.0.2 Prétraitement de la surface

— **eliminate_degenerate_faces** : Cette fonction supprime les éventuelles faces dégénérées de notre maillage, ce qui pourrait potentiellement poser problème dans la suite de notre application. Elle est appelée juste après **prepare**.

— **identify_singular_edges** : Cette fonction est appelée avant l'application du *cutting* (quelque soit le type de *cutting* choisi). En utilisant les propriétés précédemment ajoutées par la fonction **prepare**, elle va marquer les éventuelles **arêtes singulières** du maillage, marquage nécessaire pour le *cutting* qui s'ensuit.

— **identify_isolated_singular_vertices** : Cette fonction n'est appelée que dans le cas du *local cutting*. Là encore, cette étape d'identification va utiliser les propriétés précédemment rajoutées par la fonction **prepare** pour marquer les **sommets singuliers isolés** de la surface, afin que le *local cutting* s'effectue correctement.

— **eliminate_degenerate_faces** : Cette fonction supprime les éventuelles **degenerate faces** du maillage, étape nécessaire pour la suite sous peine d'avoir des résultats incohérents.

— **merge_groups** : Cette fonction est utilisée pendant la phase d'identification des **sommets singuliers isolés** via la fonction : **identify_isolated_singular_vertices**. Pour cette identification, il est nécessaire de regarder si les faces adjacentes à un sommet sont toutes *liées* entre elles. En partant d'une face adjacente arbitraire à un sommet, cette fonction permet de savoir s'il existe parmi les autres faces adjacentes une ou deux autres faces *liée* à cette autre face. Si c'est le cas, cette fonction les *rassemble*. Si l'on n'arrive pas à rassembler toutes les faces de cette manière, on peut en déduire que ce sommet est **singulier isolé**.

— **is_in_groups** : Cette fonction est elle aussi utilisée pendant la phase d'identification des **sommets singuliers isolés** (via la fonction **identify_isolated_singular_vertices**). Cette fonction est utilisée au sein de la fonction **merge_groups** et permet de savoir si une face est *liée* à d'autres faces (si elles partagent au moins une arête commune).

— **find** : Cette fonction est utilisée pendant l'identification des **arêtes singulières** et récupère le nombre de faces incidentes à une arête à partir

de ses extrémités (en comptant bien entendu les faces **non manifold** qui n'ont pas pû être ajoutées).

3.0.3 Cutting et stitching

- **local_cutting** : Cette fonction applique le *local cutting* sur le maillage d'entrée. Elle est appelée après l'identification des **arêtes singulières** et des **sommets singuliers isolés**.
- **global_cutting** : Cette fonction applique le *global cutting* sur le maillage d'entrée. Elle est appelée après l'identification des **arêtes singulières**.
- **snapping** : Cette fonction applique le *snapping* sur le maillage d'entrée. Elle est appelée après l'étape de *cutting*.
- **pinching** : Cette fonction applique le *pinching* sur le maillage d'entrée. Elle est appelée après l'étape de *cutting*.
- **pinch** : Cette fonction performe un stitch de type pinching sur le sommet pivot donné en entrée.
- **find_pivot** : Cette fonction retourne un pointeur vers un sommet pivot du maillage.
- **gather_faces** : Cette fonction rassemble les faces qui ont été découpées pendant la première étape du cutting global.
- **change_vhandle** : Cette fonction remplace toutes les occurences d'un *VertxHandle* par un autre qui a les mêmes coordonnées.

3.0.4 Détection de méthode

- **detect_cutting_type** : Cette fonction est appelée dans le cas où l'utilisateur ne précise pas le type de *cutting* voulu. Étant donné qu'il existe des différences de performance entre les différentes approches pour le *cutting*, cette fonction a pour but de détecter la meilleure à adopter selon le maillage donné en entrée par l'utilisateur.
- **detect_stitching_type** : Tout comme pour **detect_cutting_type**, cette fonction avait pour but de détecter automatiquement l'approche optimale pour l'étape de *stitching*, selon le maillage d'entrée, dans le cas où l'utilisateur ne préciserait pas le type de *stitching* souhaité.

4 Réalisations

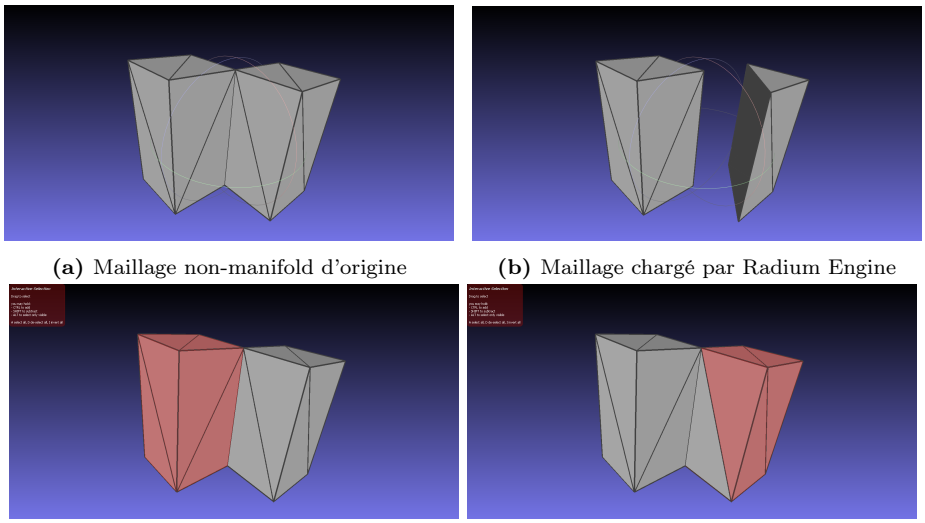
4.1 Fonctionnalités

Tâches	Implémentation	Étape	Résultat
Importation d'une surface d'entrée	Oui	Terminée	Fonctionnel
Exportation d'une surface traitée	Oui	Terminée	Fonctionnel
Elimination des degenerate faces (Prétraitement)	Oui	Terminée	Fonctionnel
Identification des arêtes singulières (Prétraitement)	Oui	Terminée	Fonctionnel
Cutting Global	Oui	Correction / Débogage	Non Fonctionnel
Cutting Local	Oui	En cours	Non Fonctionnel
Identification des sommets singuliers isolés	Oui	Terminée	Fonctionnel
Stitching : Pinching	Oui	En attente du cutting pour d'éventuelles nouvelles corrections	Fonctionnel sous réserve
Stitching : Snapping	Oui	En cours	Non Fonctionnel

4.1.1 Cutting Global

Le cutting global fait partie des 2 méthodes de cutting proposées dans l'article. Elle opère en deux temps, premièrement on déconnecte toutes les faces de la surface, deuxièmement on reconnecte en évitant les arêtes singulières. Avant d'exécuter le cutting, il faut appeler les fonctions de prétraitement associées : **eliminate_degenerate_faces** et **identify_singular_edges**.

Le cutting global que nous avons implémenté ne donne pas les résultats attendus, ceci est lié au fait que Radium Engine ne charge pas les faces non-manifold : l'algorithme présenté dans l'article itère sur les arêtes du maillage non-manifold. Puisque Radium Engine élimine certaines faces, nous perdons de l'information sur le maillage. Même si nous pouvons garder un apercu de ces faces en utilisant le foncteur **NonManifoldFaceCommand**, itérer sur le maillage importé ne peut pas fonctionner. Nous avons partiellement contourné le problème mais il reste quelques cas problématiques, nous allons en illustrer un maintenant.



(a) Maillage non-manifold d'origine

(b) Maillage chargé par Radium Engine

(c) Maillage après le cutting

(d) Maillage après le cutting

Figure 1 – Déroulement du cutting global sur 2_rectangles.obj (présent sur le dépôt)

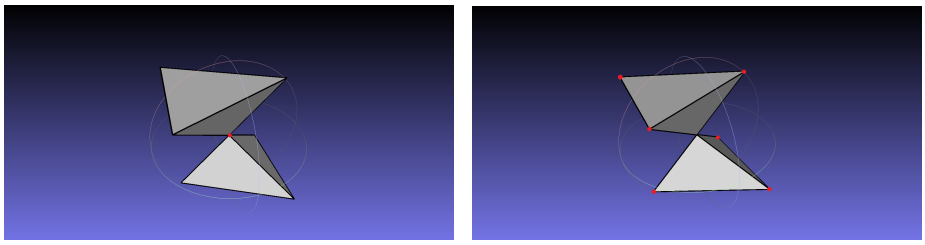
Sur la figure **1a**, on peut voir un maillage simple qui présente une arête singulière. La figure **1b** montre ce que charge Radium Engine : on voit que les faces irrégulières ont été supprimées (selon l'ordre des faces, différents résultats peuvent être observés). Quand on regarde le résultat de notre cutting global (**1c** et **1d**), on voit que les faces qui ont été enlevées par Radium Engine sont de nouveaux présentes mais le parallélépipède de droite présente un défaut topologique : les deux faces qui touchent l'arête singulière restent détachées du reste du maillage.

4.1.2 Cutting Local

Contrairement au cutting global, le cutting local découpe seulement au niveau des arrêtes marquées singulières que nous avons vu précedemment et sur les **sommets singuliers isolés**. Le marquage des **sommets singuliers isolés** doit être réalisé avant le cutting local et correspond à la fonction **identify_isolated_singular_vertices**. Ce prétraitement est fonctionnel. Cependant, comme le spécifie le tableau des réalisations, le cutting local en est encore au stade d'implémentation et n'est pas fonctionnel à ce jour.

4.1.3 Stitching Pinching

Le pinching est une des méthodes de stitching présentées dans l'article. C'est donc une étape qui vient après le cutting. L'idée de cette méthode est de chercher un sommet "pivot" sur des paires d'arêtes. Une fois qu'une paire est trouvée, on va rassembler les sommets qui se situent à l'opposé du pivot.



(a) Avant le cutting : 7 sommets (le sommet en rouge est singulier)

(b) Après le cutting : 8 sommets (les sommets en rouge sont les pivots)

Figure 2 – Résultat du pinching sur simple.obj (présent sur le repo)

4.1.4 Stitching Snapping

Le snapping est la deuxième méthode de stitching de l'article. Cette méthode permet de minimiser le nombre de composantes connexes tout en garantissant le caractère manifold de la surface. Elle permet également de coudre des arêtes qui sont localement proches (ce que le pinching ne peut pas faire). Nous avons commencé à implémenter cette méthode mais il ne s'agit que de prémices.

4.2 Difficultés

4.2.1 Radium Engine

L'une des difficultés majeure que nous avons rencontrée est la prise en main de Radium Engine. En effet, certains membres de notre groupe qui ont un système d'exploitation Windows ont eu beaucoup de mal à configurer Radium Engine afin de pouvoir l'utiliser par la suite. Il nous a fallu également comprendre comment celui-ci marchait et son architecture interne. Ainsi, il nous a fallu repenser notre manière d'implémenter. Par exemple, nous devons créer un **TriangleMesh** pour avoir un **TopologicalMesh**. Nous avons également demandé des modifications au sein de Radium Engine lui-même. Pour cela, il nous a fallu réaliser des issues sur le github de Radium Engine sous les conseils de notre encadrant. Ces demandes de modifications qui ont été réalisées de manière asynchrone nous ont pris ainsi un certain temps afin qu'elles soient adoptés et modifiés sur Radium Engine. Également, à chaque modification, une ré-installation et/ou compilation pouvait amener à des erreurs qu'il nous fallait résoudre. Nous avons notamment demandé une modification au sein de Radium Engine pour pouvoir importer une surface **non manifold**, qui est le coeur de notre projet, mais aussi par exemple le fait de pouvoir accéder à des propriétés qui ne nous était pas permise par une construction par défaut par exemple.

4.2.2 Algorithmie

Nous avons également eu des difficultés d'un point de vue algorithmique. En effet, le projet nous a été assez difficile à diviser en tâches indépendantes

et individuelles puisque le principe même de notre projet repose sur un modèle séquentiel. Par exemple, pour pouvoir effectuer nos tests de manière poussée et détaillée sur le stitching, le cutting doit se réaliser avant celui-ci puisqu'il prend en entrée le résultat du cutting. Également, pour pouvoir compléter notre algorithme sur les fonctions comme le cutting, les prétraitements se devaient d'être terminer et fonctionnels. Dans notre cas, nous avons pris un retard sur le cutting qui est à ce jour non fonctionnel puisque nous sommes encore en train de corriger des erreurs et des bugs. Le stitching qui est fonctionnel d'après les tests que nous avons effectué sur celui-ci est en réalité en attente de la terminaison du cutting puisqu'il ne sera validé qu'une fois avoir été testé avec les résultats du cutting fonctionnel. Nous ne pouvons pas le considérer comme fonctionnel, et est à ce jour non fonctionnel pour cette raison.

4.2.3 Communication

Pour finir, nous avons eu une dernière difficulté majeure qui n'est autre que la communication au sein du groupe. Dans le contexte de crise sanitaire que le monde traverse, il est difficile de pouvoir se réunir physiquement pour parler et visualiser le projet ensemble. Pour cette raison également, certains membres du groupe ne sont pas proches du campus universitaire et sont retournés auprès de leur famille. Cette option ne semblait donc pas envisageable. Un lien purement distanciel joue sûrement dans ce manque de communication.

Une deuxième raison de ce manque de communication réside dans le manque d'implication que nous avons eu sur la transmission et le partage de notre avancement au fur et à mesure des jours. Nous avions décidé d'utiliser l'outil **Trello** pour une visualisation globale des tâches à effectuer et l'avancement du projet dans sa totalité. Cet outil a au fur et à mesure été délaissé et a donc contribué à un manque de communication au sein du groupe couplé avec sans nul doute une perte de temps.

4.3 Risques

Risques	**Prévu**	**Rencontré**	**Impact initial**	**Impact réel**	**Solutions initiales / Inconvénients**
Prise en main difficile de Radium et OpenMesh	Oui	Oui	Important	Très important	Questions à l'encadrant et issues sur github / Très long en asynchrone
Retard sur le planning et circonstances inattendues	Oui	Oui	Moyen	Important	Organisation prévoyante et réaction rapide / Difficile à distance
Mauvaise répartition des tâches et manque de communication	Oui	Oui	Moyen	Important	Mise en commun et adaptabilité des membres à aider un autre membre sur une tâche / Peu réalisé, difficile de corriger le code de quelqu'un d'autre
Abandon d'un membre du projet	Oui	Non	Important	Aucun	
Fonctions de prétraitement non fonctionnelles	Oui	Non	Important	Aucun	
Perte de données	Oui	Non	Moyen	Aucun	

5 Améliorations

Comme expliqué précédemment, la première partie correspondant aux techniques de cutting global et de stitching pinching sont quasiment fonctionnelles. Pour des maillages simples, celles-ci fonctionnent correctement. Les améliorations possibles sur notre projet sont donc premièrement une correction légère sur cette première partie pour une finalisation et un résultat satisfaisant afin d'obtenir une surface manifold pour tout type de surface d'entrée.

Une seconde amélioration serait la finalisation de notre deuxième partie d'implémentation correspondant aux techniques de cutting local et de stitching snapping afin de pouvoir choisir une approche et une méthode spécifique à appliquer à une surface d'entrée.

Au final, suite aux nombreuses petites difficultés que nous avons rencontrées, nous avons abandonné nos fonctionnalités optionnelles que nous avions identifiées qui pourrait faciliter et améliorer l'utilisation de notre projet. Ces fonc-

tionnalités améliorant le confort d'utilisation pour l'utilisateur sont également des améliorations possibles pour notre projet. Nous avions identifié quelques fonctionnalités optionnelles spécifiées dans nos précédents rapports comme la reconnaissance des méthodes qui maximisent l'efficacité selon une surface d'entrée spécifique donnée, ou encore comme la visualisation des surfaces résultantes.

Dans une autre possibilité, les fonctionnalités apportées par ce projet pour une correction topologique d'une surface d'entrée peuvent être reprises dans une application ultérieure. Nous nous sommes efforcés de produire une implémentation modulable et intégrable pour cette possibilité.

6 Conclusion

Pour conclure, comme spécifié précédemment, le résultat de nos implémentations n'est pas encore suffisant pour correspondre aux méthodes présentées dans notre papier de référence de GUEZIEC et al. [2001]. Cependant, le projet tel qu'il est corrige tout de même les singularités topologiques et de ce fait, les résultats de l'application sont utilisables (même si différents des résultats théoriques attendus). Une légère amélioration sur le cutting global et une poursuite des tests après cette amélioration sur le stitching pinching permettraient un résultat final pour n'importe quel type de surface d'entrée.

Les difficultés rencontrées et évoquées se sont accumulées et sont sans nul doute responsables de l'absence d'un résultat finalisé. Malgré cela, nous nous sommes confrontés à ces difficultés et avons appris à les résoudre. Ce projet nous a également permis d'en apprendre davantage sur le fonctionnement d'OpenMesh mais aussi de s'essayer sur l'implémentation de méthodes évoquées dans un papier de recherche scientifique et notamment sur le sujet de la géométrie des maillages qui nous tenait tous à coeur.

Glossaire

arête singulière Une arête est dite *singulière* si au moins trois faces lui sont incidentes. Autrement, on parle d'arête *régulière*. 6, 7, 9, 14

degenerate face On parle de *degenerate face* lorsqu'une face possède au moins deux sommets identiques. 6, 8

manifold Une surface polygonale est dite *manifold* si tous ses sommets sont réguliers. Dans l'autre cas, on parle d'une surface *non manifold*. 1

non manifold Une surface polygonale dite *non manifold* si au moins un de ses sommets est singulier. Dans l'autre cas, on parle d'une surface *manifold*. 1, 4, 5, 7, 10

sommet singulier Un sommet v est dit *singulier* si $l(v)$ n'est ni *chaîne*, ni un *cycle*. Autrement, on parle de sommet *régulier*. 9, 14

sommet singulier isolé Un sommet *singulier isolé* correspond à un **sommet singulier** qui n'est pas une extrémité d'une **arête singulière**. 6, 7, 9

Références

Marco ATTENE, Marcel CAMPEN, and Leif KOBBELT. Polygon mesh repairing : An application perspective. *IMATI-GE Consiglio Nazionale delle Ricerche and RWTH Aachen University*, 2013.

André GUEZIEC, Gabriel TAUBIN, Francis LAZARUS, and Bill HORN. Cutting and stitching : Converting sets of polygons to manifold surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 7(2), 2001.