Université de Bourgogne
M2 Image et Intelligence Artificielle

11 janvier 2021
Rapport de stage

Complétion de modèles numériques 3D en immersion virtuelle

Julien Brissonnet
Encadré par :
Fabrice Jaillet
Jean-Rémy Chardonnet

Remerciements

Je souhaite remercier toutes les personnes qui ont contribué au bon déroulement de mon stage et ont permis l'avancée de mon projet.

Je remercie particulièrement mes tuteurs de stage, Fabrice Jaillet et Jean-Rémy Chardonnet, pour leurs conseils, pour leur suivi tout au long du stage, ainsi que pour leurs encouragements et pour la confiance qu'ils m'ont accordé.

Je voudrais ensuite remercier mes professeurs, et en particulier Christian Gentil pour son accompagnement dans notre recherche de stage et Cyrille Migniot pour être venu sur mon lieu de stage s'assurer que tout se passait bien.

Je remercie également tout le personnel de l'institut image pour son accueil et sa sympathie. Les ingénieurs et techniciens qui m'ont aidé lorsque j'ai été confronté à des problèmes techniques, et particulièrement Julien Ryard qui a pris du temps pour m'aider à résoudre des problèmes et m'expliquer certaines notions à plusieurs reprises.

Sommaire

1 Introduction

La numérisation 3D consiste à mesurer les formes et les dimensions dans l'espace d'un objet, à l'aide de systèmes d'acquisition divers et variés suivant les objectifs poursuivis, afin de transformer ces informations en un fichier exploitable par un ordinateur. Pour de nombreuses applications en informatique, notamment dans les domaines de la simulation, du jeux vidéo et des effets spéciaux, nous souhaitons créer un modèle géométrique en 3D de ces objets à partir des informations issues de l'acquisition. Une des représentations les plus courantes des objets en 3D est le maillage surfacique, et c'est celle que nous utiliserons dans ces travaux.

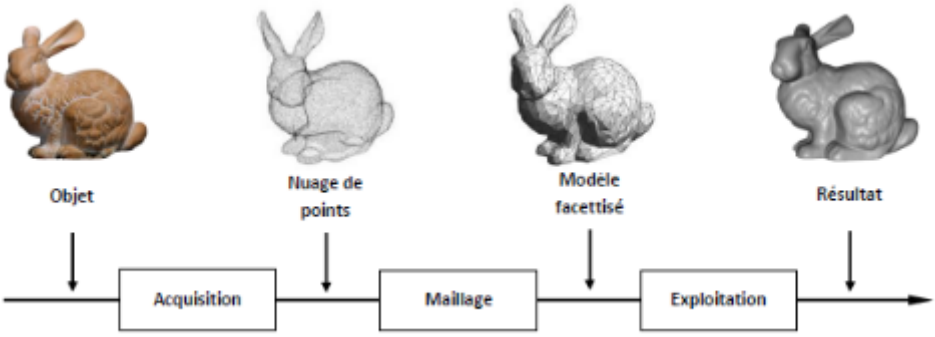


FIGURE 1 – Chaine de traitement de numérisation 3D

En fonction du contexte et des conditions de l'acquisition, il est fréquent que les informations obtenues par les mesures soient incomplètes, notamment si certaines parties de l'objet à scanner sont inaccessibles, ou même inexistantes (imaginons par exemple que l'on souhaite restaurer virtuellement un objet qui aurait été détérioré). Les maillages générés par ces acquisitions seront donc également incomplets. Si ces zones incomplètes restent petites, il reste relativement facile de les combler automatiquement par des méthodes de remplissage et de lissage [1-2]. Cependant lorsque des parties significatives de l'objet sont manquantes, il faudra utiliser d'autres méthodes, définir des contraintes pour la complétion, basées sur l'étude des caractéristiques du modèle. Nous allons donc nous intéresser à concevoir un outil logiciel permettant à un utilisateur de reconstruire un maillage en extrapolant les parties manquantes à l'aide de contraintes qu'il aura lui-même définies de manière aussi intuitive que possible. Une particularité de cet outil est qu'il exploitera l'immersion virtuelle afin de donner à l'utilisateur une perception accrue et une manipulation facilitée du modèle qu'il vise à reconstruire, ce qui pourrait l'aider dans sa tâche. Nous nous intéresserons plus particulièrement à l'exploitation de dispositifs de visualisation en 3D et de retour de force.

2 Contexte

2.1 Déroulement du stage

Ce stage a été co-encadré par Fabrice Jaillet, enseignant-chercheur du laboratoire Liris à Lyon et Jean-Rémy Chardonnet, enseignant-chercheur de l'Institut Image à Chalon-Sur-Saône.

Le Liris, Laboratoire d'InfoRmatique en Image et Systèmes d'information, est une unité mixte de recherche dont les tutelles sont le CNRS, l'INSA Lyon, l'Université Claude Bernard Lyon 1, l'Université Lumière Lyon 2 et l'École Centrale de Lyon. Il compte environ 320 membres, répartis en 14 équipes de recherche structurées autour de 6 pôles de compétence, comme le montre la figure 2 ci-dessous.

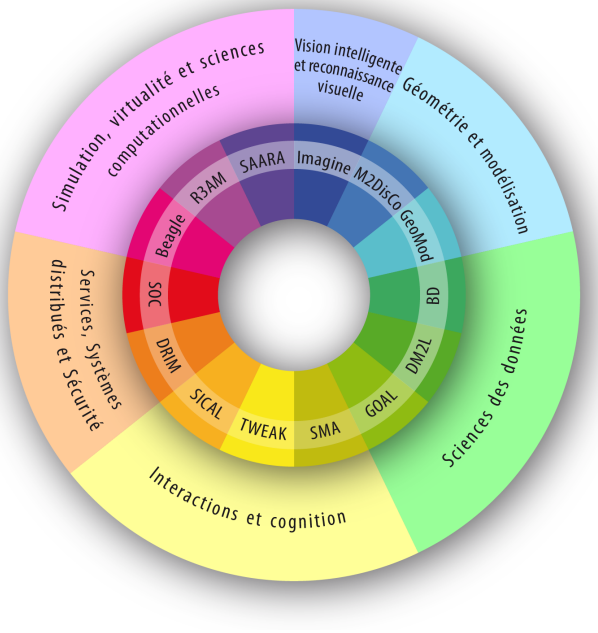


FIGURE 2 – Poles et équipes du Liris

Fabrice Jaillet est membre de l'équipe SAARA : Simulation, Analysis and Animation for Augmented Reality appartenant au pôle Simulation, virtualité et sciences computationnelles. Comme nous le verrons dans la section suivante, le sujet de ce stage est initialement né parallèlement à un projet mené par cette équipe.

L'institut Image est un institut d'Arts et Métiers ParisTech intégré au centre de Cluny, dédié à la recherche et à l'innovation dans le domaine de l'immersion virtuelle, ainsi qu'à la formation puisqu'il accueille deux promotions de Master. Il regroupe un effectif d'environ 25 personnes, composé d'enseignants-chercheurs, de doctorants, de post-doctorants, de techniciens et d'ingénieurs. Son équipe de recherche est l'une des équipes du Laboratoire d'Electronique, Informatique et Image (LE2i) affilié à l'Université de Bourgogne, aux Arts et Métiers et au CNRS. C'est dans les locaux de l'institut à Chalon-sur-Saône que j'ai effectué mon stage, afin de bénéficier du matériel et de leur expertise dans le domaine de l'immersion virtuelle.

2.2 Sujet

Ce stage n'était pas le premier à traiter cette problématique, même si cette dernière a évolué au cours du temps comme nous allons le voir maintenant. Un premier stage avait été réalisé en 2011 par Zhifan Jiang [16]. Initialement, la problématique est née de la volonté de créer des modèles géométriques d'organes à partir de données issues d'appareils d'imagerie médicale pour une application de simulation d'accouchement développée par l'équipe SAARA du Liris. Souvent ces données ne décrivent qu'une partie de l'organe observé et le but était alors de reconstruire les parties manquantes afin d'obtenir des organes au volume réaliste. Dans ce cadre, la solution retenue a été la suivante :

— En premier lieu, une décomposition du problème en trois étapes : triangulation, raffinement et lissage, suivant la méthode décrite par P. Liepa dans [1], qui prend en compte les caractéristiques du modèle aux frontières du trou pour la reconstruction. Cette procédure est illustrée ci-dessous (Fig. 3) ;

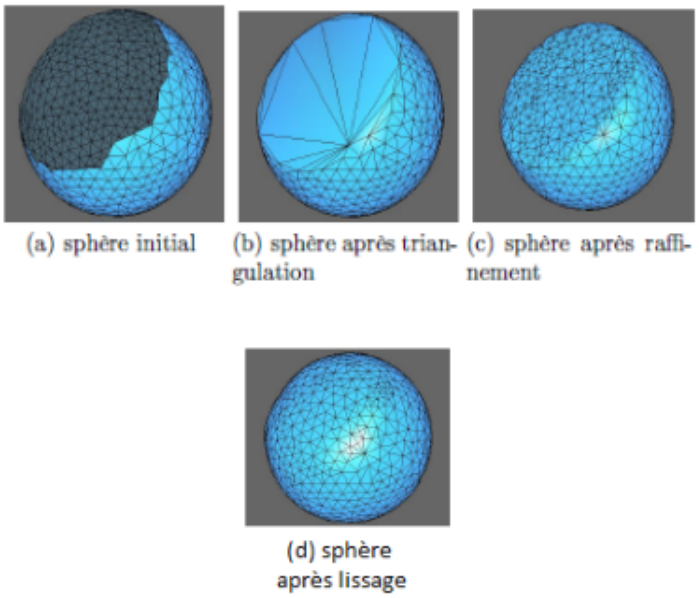


FIGURE 3 – triangulation, raffinement et lissage sur une sphère trouée. Résultats issus de [16]

— Puis l'ajout d'une contrainte de position, servant de base à une déformation du maillage obtenu à la fin de la procédure précédente. Si l'on se réfère à la sphère obtenue dans l'exemple ci-dessus, elle présente une continuité de la courbure aux frontières, mais nous avons envie de tirer sur le patch reconstruit pour lui donner une véritable forme de sphère. La solution retenue pour effectuer cette déformation repose sur une méthode dite linéaire car elle repose sur la résolution d'un système linéaire, décrite par Botsch et Sorkine dans [3].
Comme nous le verrons par la suite, cette chaîne de traitement initiale est très proche de celle que nous utilisons dans notre solution.
La problématique a ensuite été reprise en 2014 par Armand Le Gouguec [17], lors d'un stage qui visait à faire un état des lieux plus exhaustif des différentes techniques existantes pour la complétion de trous, afin de mettre en place une méthode plus souple, et d'élargir la problématique en ne se limitant plus à des complétions d'organes mais en envisageant d'autres applications comme la rétro-conception dans le domaine du patrimoine, de l'ingénierie mécanique.. Autrement dit, la solution devrait permettre de traiter à la fois des modèles présentant des formes arrondies, et d'autres contenant plus d'arêtes vives, aux contours plus nets, avec plus ou moins de détails à reconstituer. Plusieurs éléments sont ressortis de ces travaux :
— D'abord, qu'il serait intéressant de travailler non pas directement sur le modèle 3D de l'objet, défini par un grand nombre de polygones, mais sur un modèle simplifié, qui serait construit à partir d'éléments caractéristiques du maillage 3D initial, sans pour autant perdre énormément d'informations sur sa géométrie. Ces éléments peuvent être des arêtes vives, des sommets caractéristiques, des lignes de courbures...
— À partir de l'analyse sémantique de ce modèle simplifié, l'idée est de définir un certain nombre de contraintes pour guider la reconstruction des patchs dans les zones où le maillage est endommagé. A cette fin, nous pourrons exploiter par exemple les symétries, ou la forme aux frontières des éléments caractéristiques du modèle.

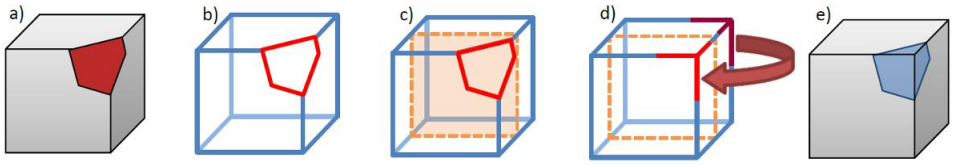


FIGURE 4 – flux de traitement de la méthode proposée, image issue de [17].

— Les travaux décrivent en particulier l'utilisation des lignes de courbure extrémales (également appelées ridges) comme éléments caractéristiques pour construire le modèle simplifié. Nous reviendrons par la suite sur ces lignes de courbure extrémales et sur la pertinence de leur utilisation. Mais nous pouvons déjà expliquer que l'intérêt pour ces dernières tient au fait qu'elles sont capables de rendre compte des formes anguleuses comme des formes arrondies.

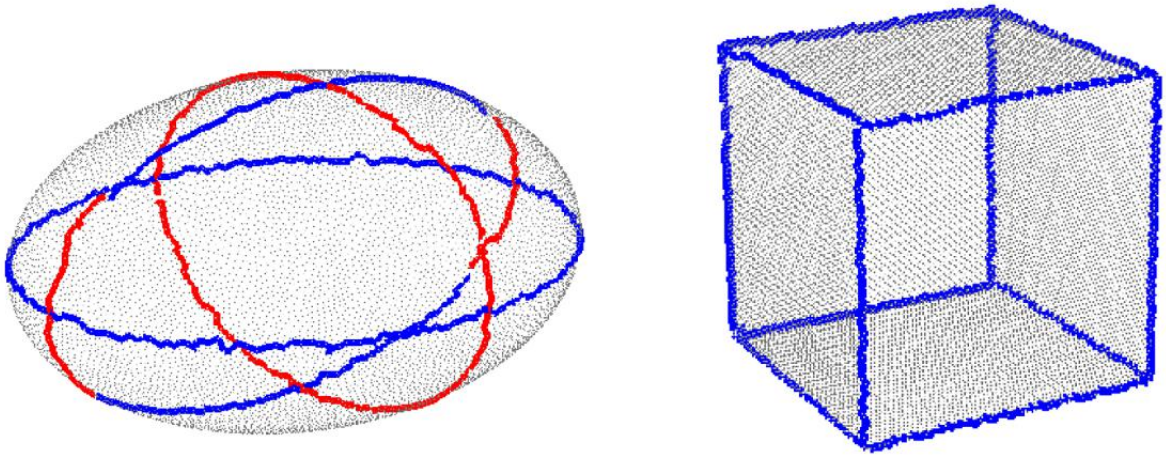


FIGURE 5 – résultat du calcul des ridges sur un cube et une ellipsoïde, résultats issus de [17].

— Enfin, ces travaux ont permis indirectement de mettre en évidence le fait qu'il n'est pas toujours aisé de détecter automatiquement les informations nécessaires à la reconstruction du modèle... Par exemple : quels sont les lignes/formes et les symétries à considérer pour reconstruire la zone du maillage endommagée qui nous intéresse ? Ces opérations peuvent être très compliquées à mettre en place automatiquement, c'est pourquoi la réflexion autour d'une interface utilisateur s'est imposée.
Tout ce travail préalable nous a donc servi de base afin de définir un cap à suivre et de fixer nos objectifs durant ce stage. De manière plus générale, nous précisons également que dans le cadre de ce projet nous travaillerons sur des maillages triangulaires déjà construits et qui présentent des zones trouées de taille plus ou moins importante et non pas directement sur des nuages de points.

3 Objectifs

Si l'on résume les éléments évoqués lors de la description du sujet et des travaux précédents, le projet de ce stage va donc s'articuler autour de deux principaux objectifs :

— En premier lieu la création d'un outil logiciel interactif exploitant la détection des ridges sur un maillage afin de les utiliser pour la reconstruction des parties détériorées de ce dernier. Il faudra déterminer les modalités d'exploitation et de manipulation des ridges, implémenter les méthodes présentées précédemment afin d'obtenir un prototype nous permettant d'évaluer les techniques envisagées. Nous évaluerons notamment la pertinence de l'utilisation des ridges pour réaliser le modèle simplifié de nos maillages, et contraindre la reconstruction.

— Puis la réflexion autour de modalités d'interaction en immersion virtuelle, en utilisant notamment des dispositifs de visualisation en 3D et de retour de force, et l'évaluation de l'apport de l'immersion virtuelle dans le cadre de la reconstruction de maillage. Ces interactions devront bien sûr pouvoir être réalisées en temps réel, et nous nous efforcerons de les rendre les plus simples et intuitives possible.

Bien sûr ces deux objectifs n'ont pas été traités de manière totalement indépendante et se sont bien souvent entrecroisés, puisque les solutions que nous avons retenues tout au long du projet pour interagir avec le maillage et les ridges ont toujours été choisies en gardant en tête le contexte d'utilisation finale en immersion virtuelle.

On peut également noter qu'il n'y avait pas de cahier des charges précisément défini étant donné que le travail avait une portée assez exploratoire, dans le sens où nous n'avions pas initialement une idée fixée de la manière d'exploiter les ridges comme contrainte ou de faire intervenir la réalité virtuelle, et je disposais d'une liberté relativement importante pour répondre au mieux aux objectifs de la problématique. Aussi, j'ai opté dans un premier temps pour les solutions les plus simples afin d'avoir un premier prototype opérationnel à la fin du stage.

Un premier travail préparatoire a été de réfléchir à l'architecture du système que nous souhaitions mettre en place, aussi bien sur le plan matériel que logiciel. Dans la partie suivante, nous allons expliquer ce premier travail et les choix réalisés en fonction des outils à notre disposition.

4 Architecture du projet

4.1 Environnement matériel

Dès le départ nous avions en tête l'idée d'être capable de toucher, de tourner autour de l'objet à reconstruire. Initialement, il était prévu d'implémenter le logiciel de reconstruction sur un dispositif SPIDAR.

Le SPIDAR est un dispositif de retour de force basé sur l'utilisation de fils reliés à des moteurs, offrant six degrés de liberté. Un tel dispositif aurait dû être disponible à l'institut Image, installé dans une salle immersive composée de quatre écrans (un au sol, un en face et deux sur les côtés) permettant une visualisation en 3D stéréoscopique. Mais son installation a pris du retard, c'est pourquoi nous avons envisagé une solution alternative utilisant un casque de réalité virtuelle et un bras articulé haptique.

FIGURE 6 – casque de réalité virtuelle HTC Vive et bras haptique Phantom Omni

Nous avons à disposition à l'institut Image un casque HTC Vive. La particularité de ce dernier est qu'il est fourni avec des contrôleurs et un système de tracking composé de deux caméras délimitant une zone dans laquelle la position du casque et des contrôleurs est trackée et dans laquelle on peut se déplacer. Cependant, pour notre application nous n'exploiterons à priori pas cette possibilité puisque nos déplacement seront limités par la position et l'amplitude du bras haptique.

L'atout majeur du dispositif Spidar par rapport à cette solution est qu'il permet l'interaction dans un environnement immersif à échelle humaine, cependant, pour le type d'application de manipulation de maillage que nos souhaitons réaliser, cet avantage ne représente pas une contrainte critique et il est tout à fait possible d'imaginer une solution dans un espace plus restreint, correspondant à un poste de travail sur lequel est placé le bras haptique par exemple. De plus, l'objectif est que la solution soit facilement adaptable ensuite pour le dispositif Spidar. Et cela devrait être le cas car comme nous allons le voir plus en détail dans la section suivante, les deux solutions utilisent le même logiciel pour réaliser l'interface avec le matériel de réalité virtuelle et créer les environnements virtuels immersifs : le moteur de jeu Unity3D.

4.2 Environnement Logiciel

4.2.1 Unity3D

Afin de réaliser des projets de réalité virtuelle/réalité augmentée à l'Institut Image, le logiciel principalement utilisé est Unity3D. C'est donc assez naturellement que l'on s'est tourné vers cette plateforme pour programmer les différentes modalités de manipulation et pour gérer la communication avec le matériel de réalité virtuelle. Ce dernier aspect est d'ailleurs quasiment transparent comme nous allons le voir.

Unity 3D est un moteur de jeu permettant de créer de environnements virtuels. Il présente quelques caractéristiques très intéressantes pour la réalisation d'applications 3D en temps réel, comme son propre moteur physique, son système de collisions, son moteur d'ombre et de lumières... Il permet de plus d'exporter nos projets vers de multiples plateformes, mobiles, web, windows, mac,

consoles de salons et consoles portables... Il est notamment compatible nativement avec plusieurs casques de réalité virtuelle dont le HTC Vive que nous utiliserons.

Un projet Unity est composé d'un ensemble d'éléments appelés "Assets". Les Assets correspondent à n'importe quel type d'éléments qui peut être utilisé dans un projet Unity. Ils peuvent aussi bien provenir de fichiers crées en dehors d'unity, comme des modèles 3D crées avec notre modeleur 3D favori, des fichiers son ou image, que de fichiers directement crées avec Unity, comme les "materials" permettant de créer des matériaux que l'on pourra attribuer aux objets d'une scène, les scripts C#, ou justement ce que l'on appelle les scènes en elles-mêmes, et qui correspondent à l'environnement virtuel que l'on a créé et sont elles-mêmes des assets du projet. On peut également télécharger des "packages" sur l'"Assets Store" d'Unity, qui correspondent à un ensemble d'Assets créés par la communauté ou par des professionnels et que l'on souhaite utiliser dans notre projet.

Parmi ces packages, nous nous intéresserons particulièrement au steamVR plugin, développé par Valve (développeur du HTC Vive) qui nous fournit gratuitement une API permettant de gérer le système composé du casque de réalité virtuelle et de ses différents périphériques (position et utilisation des contrôleurs..., etc.) directement dans Unity et de manière assez intuitive.

Nous avons également trouvé un package fournissant une API pour utiliser directement le Phantom Omni dans Unity. Il s'apelle "Haptic Plugin for Geomagic OpenHaptics 3.3" et a été développé par "School of Simulation and Visualisation (SimVis), Glasgow School of Art". Il réalise une interface entre Unity et Open Haptics toolkits 3.3, un ensemble d'outils permettant de gérer le bras et les fonctionnalités de retour de force haptique.

Enfin, tous les objets dans une scène d'un projet Unity sont appellés GameObjects. Ce sont les objets fondamentaux d'une scène réalisée avec Unity, et on peut les considérer comme de gros conteneurs. Ils peuvent correspondre aussi bien à un élément de l'environnement, un personnage jouable, qu'à une source de lumière, une caméra, etc... on leur attribue des composants, qui peuvent être de natures diverses : par exemple un MeshFilter permet de stocker un maillage pour le GameObject pour initialiser sa géométrie, un Collider permet de gérer les collisions de cet objet, on peut également lui attribuer un comportement personnalisé en lui attachant un ou plusieurs scripts écrits en C# (ou UnityScript, langage proche du JavaScript). L'annexe A illustre l'organisation de ces différents éléments telle qu'elle est représentée à l'intérieur de l'environnement de travail d'Unity3D.

4.2.2 CGAL

Cependant, nous n'allons pas pouvoir réaliser toutes les opérations de notre chaîne de traitement directement sur Unity. En effet , cette dernière comprend plusieurs opérations sur les maillages plutôt complexes : le calcul des ridges, la complétion des trous et les opérations qu'elle entraîne, raffinement, lissage et déformation du patch (procédure décrite à la figure 3). Il est possible de manipuler les maillages directement sur Unity, cependant la structure de données qui les décrit est extrêmement basique puisqu'il ne s'agit que d'une liste de sommets et de facettes.

Lors des précédents stages, plusieurs briques de logiciel avaient déjà été réalisées sur la plateforme Mepp. Mepp est un framework développé par le Liris et le CNRS, qui permet de fournir des outils pour implémenter des processus de traitement de maillages et une interface utilisateur basée sur Qt pour pouvoir les tester. Dans notre solution, nous n'utiliserons pas de projet Mepp, car la partie visualisation et interface utilisateur sera gérée par Unity. Néanmoins, pour effectuer ces opérations de manipulation sur les maillages, Mepp s'appuie sur la librairie CGAL.

CGAL pour Computationnal Geometry Algorithm Library est une librairie écrite en C++, développée collaborativement par plusieurs universités, instituts de recherche et entreprises et qui met à disposition des structures de données et des algorithmes géométriques fiables et efficaces. La classe principalement utilisée par Mepp et qui nous intéressera pour notre solution est la classe "Polyhedron_3" de CGAL, qui permet de décrire un maillage dans l'espace en 3D. Elle est structurée de la manière suivante

- Chaque arête (edge) se décompose en 2 demi-arêtes orientées dans des sens opposés (halfedge). Chaque demi-arête possède donc une jumelle d'orientation opposée (opposite halfedge).
- Les extrémités d'une demi-arête sont des sommets (vertex). Celui vers lequel la demi-arête pointe est appelé sommet incident (incident vertex).
- Une demi-arête est précédée d'une autre demi-arête (previous halfedge). Elle est également suivie d'une autre demi-arête (next halfedge).

— Une succession de demi-arêtes forme une boucle décrivant les bords d'une facette (facet) ou d'un trou (hole).

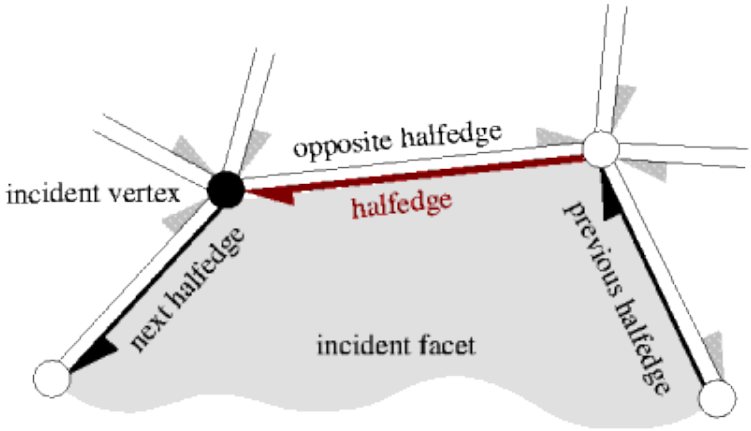


FIGURE 7 – illustration de la structure de données fournie par la classe Polyhedron_3, image issue de [4]

Il est ainsi plus facile de parcourir le maillage pour y effectuer des opérations. Par contre, cette structure de donnée impose que les maillages manipulés soient 2-Manifold. un maillage est dit 2-Manifold s'il ne contient ni arêtes non-manifolds, ni sommets non-manifolds, ni intersections. Il ne doit pas y avoir non plus de faces ouvertes ni d'éléments libres comme des arêtes ou des sommets [5].

Une arête non-manifold est une arête qui est commune à plus de deux éléments (triangles) (voir Fig. 8(c)).

Un sommet non manifold est un sommet qui est commun à plus de deux arêtes frontières (voir Fig. 8(a)(b)).

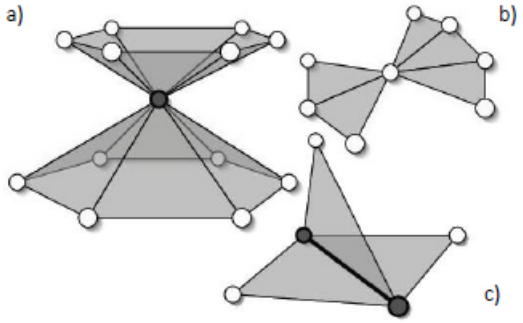


FIGURE 8 – Exemple de maillages non-manifold (image issue de [5])

Enfin, comme nous le verrons dans le chapitre suivant, plusieurs algorithmes très utiles dans le cadre de notre problématique ont été implémentés dans la nouvelle version de la librairie CGAL, alors que pour certains ils ne l'étaient pas encore au moment des premiers stages.

4.3 Solution retenue

Au vu de ces éléments, nous avons décidé que toute la partie interaction/visualisation serait gérée par Unity, la communication avec le matériel de réalité virtuelle par des Assets d'Unity déjà développés, tandis que les opérations plus complexes sur le maillage seraient codées en C++ en utilisant CGAL. Nous avons envisagé de distribuer l'application sur plusieurs machines, afin d'exécuter les traitements sur le maillage en parallèle avec les autres tâches réalisées côté Unity, ou même simplement de les exécuter en parallèle sur une même machine, mais finalement, pour plus de simplicité et pour accélérer la phase de prototypage, ces fonctionnalités seront injectées dans

l'application sous forme de Plugin. Les Plugins d'Unity 3D sont des Assets qui correspondent en fait à des librairies dynamiques. Les fonctions de ces librairies seront donc exécutées séquentiellement à leur appel, à l'exécution du programme. On précisera également que les librairies utilisées sont des dll, ce qui limite leur utilisation à un environnement Windows. Cependant Unity supporte d'autres types de librairies en tant que plugin et il est tout à fait envisageable d'étendre la compatibilité à d'autres environnements par la suite.

5 Mise en œuvre

5.1 Communication de données avec les plugins

La première difficulté posée concerne les modalités d'échange de données entre les scripts de l'application unity3D et les librairies dynamiques utilisées comme plugins. Comment échanger des données traitées dans un script C# sous Unity avec un programme écrit en C++ ? Est-ce même seulement possible ?

L'éditeur de script d'Unity est compatible Mono, une mise en œuvre open-source de la plateforme .NET de Microsoft. Comme la plateforme .NET, Mono utilise le Common Language Runtime (CLR), une machine virtuelle qui va traduire le code écrit dans le langage haut niveau de notre choix (pour nous le C#) en un langage intermédiaire (le CIL, pour Common Intermediate Language) qui pourra être ensuite compilé en langage natif de la machine. Il permet ainsi l'interopérabilité entre différents langages, et apporte d'autres fonctionnalités comme l'optimisation de la mémoire à l'exécution avec un ramasse-miettes.

Pour cette raison, Unity3D accepte deux types de plugins :

— Les plugins managés : ce sont des plugins écrits dans un langage appartenant à l'environnement .NET/Mono. Ils ont aussi été compilés en passant par l'intermédiaire du CLR, ils sont ce que Microsoft appelle des codes "managés". Leur utilisation n'est finalement pas très différente de celle des scripts C#, si ce n'est qu'ils sont compilés en dehors d'Unity3D.

— Les plugins natifs : ils sont écrits dans des langages (comme le C, C++...) directement compilés en code natif sans passer par une machine virtuelle comme dans l'environnement .NET/Mono. C'est ce genre de plugins qui nous intéresse car ils vont permettre de faire appel à une librairie tierce comme CGAL. Mais ils ne peuvent donc pas bénéficier de l'interopérabilité directe fournie par l'environnement Mono. Cependant, ce dernier facilite l'utilisation des librairies écrites en code non managé. Il est ainsi possible de charger une librairie en utilisant la fonction "DLLImport" qui va nous permettre d'invoquer les fonctions d'une librairie. Cependant, cela va poser une difficulté importante : le passage des paramètres et la récupération des données. En effet, si il est assez facile de passer des données de type simple comme des entiers ou des flottants d'un programme en C++ à un programme en C# et vice-versa, c'est bien plus difficile voire parfois impossible pour des structures ou des classes plus compliquées.

Nous reviendrons plus en détails sur les données à échanger propres à chaque manipulation du maillage dans les sections suivantes, mais on peut déjà réfléchir à l'échange entre l'application Unity3D et la Dll des données relatives au maillage que l'on souhaite manipuler.

A priori dans un premier temps, les seules informations que l'on souhaite transmettre sont la géométrie et la topologie du maillage.

Sous Unity3D, ces dernières sont représentées de la manière suivante :

— Pour la géométrie : une liste de Vector3 correspondant aux coordonnées des sommets du maillage. On rappelle que la classe Vector3 en C# est une classe permettant de manipuler des vecteurs à 3 dimensions, adaptée pour décrire des points dans l'espace donc.

— Pour la topologie : une liste d'entiers correspondant aux indices des sommets dans la liste précédente. Trois indices successifs correspondent à 3 sommets qui forment une facette triangulaire.

— d'autres informations peuvent être stockées comme les normales en chaque point...

La représentation des Maillages dans CGAL utilise la classe Polyhedron_3 évoquée au chapitre précédent, et se révèle plus subtile et donc plus complexe puisqu'on le rappelle, elle permet de stocker des informations sur les sommets, les facettes, les arêtes représentées sous forme de demi-arêtes et les relations entre ces dernières.

Pour éviter des procédures complexes et garantir l'efficacité du programme, nous avons choisi de transmettre ces informations sous forme de tableaux de flottants. La compatibilité de ces types simples est directe entre les codes managés et non managés. Pour passer le maillage en paramètre à la Dll, on construit donc un tableau de flottants qui correspond à une trame organisée de la manière suivante :

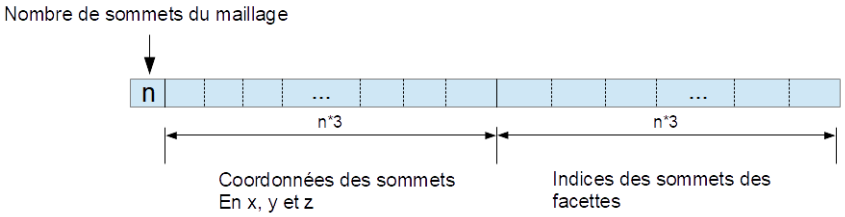


FIGURE 9 – Trames d'échange de données du maillage entre le programme principal et les dlls

Du côté de la dll, on va reconstruire à partir de cette trame un objet de la classe *Polyhedron_3* en s'aidant des outils de la classe auxiliaire *Polyhedron_incremental_builder_3*. De la même façon, lorsque l'on voudra renvoyer le maillage après l'opération réalisée par la dll au programme principal, on reconstruira une trame du côté de la dll qui correspondra à la valeur de retour de la fonction que l'on appelle depuis le code managé.

Nous allons maintenant revenir plus en détail sur la mise en œuvre de chacun des aspects de notre application, sur les choix qui ont été faits pour illustrer au mieux notre technique, sur les informations nécessaires à échanger pour chaque opération importante sur le maillage et sur le principe général des ces opérations.

5.2 Calcul des ridges sur la surface

5.2.1 Rappel sur les ridges

Nous allons revenir dans un premier temps sur cette notion de "ridges", ou lignes de courbures extrémales. La courbure d'un objet géométrique est la mesure du caractère "plus ou moins courbé" de cet objet.

Si l'on considère une courbe planaire, géométriquement la courbure en un point correspond à l'inverse du rayon du cercle osculateur à la courbe en ce point. Le cercle osculateur à une courbe en un point est le cercle passant par ce point qui épouse la courbe le mieux possible. On peut aussi définir cette grandeur comme la norme de l'accélération d'un mobile parcourant la courbe à une vitesse constante de 1.

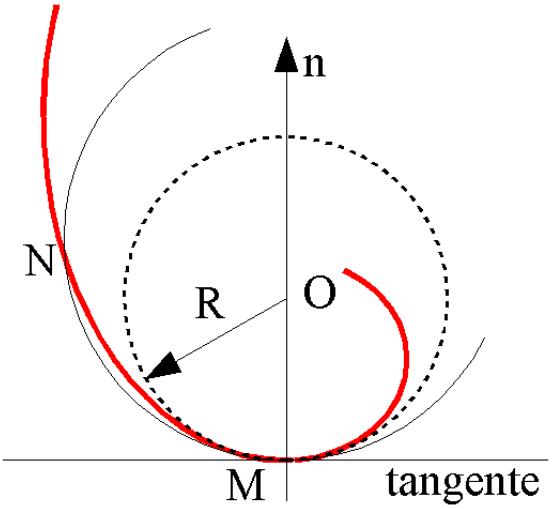


FIGURE 10 – Cercle osculateur (en pointillé) : le cercle osculateur épouse mieux la courbe qu'un cercle tangent quelconque (traits continus). Son centre O est le centre de courbure et son rayon R le rayon de courbure de la courbe en M.

Dans le cas d'une surface définie dans l'espace, la courbure n'est plus une valeur unique. On considère un plan passant par la normale à la surface au point où l'on étudie la courbure. L'intersection de ce plan avec la surface est une courbe planaire dont on va pouvoir définir la courbure

de la manière décrite ci-dessus.

Il existe une infinité de plans passant par la normale à la surface au point considéré, que l'on obtient en effectuant une rotation du plan autour de cette normale. Il en résulte donc une fonction π périodique qui associe à chaque angle de rotation du plan une valeur de courbure. Cette dernière passe par un minimum global appelé courbure principale minimale, et par un maximum global appelé courbure principale maximale.

Ces courbures principales sont associées à des plans de courbure, que l'on peut caractériser par une direction définie par l'intersection de ces plans avec le plan tangent à la surface au point considéré (voir figure 11). Ces directions sont appelées directions principales. Il est établi que la direction principale maximale et la direction principale minimale sont orthogonales. Elles forment donc avec la normale à la surface au point considéré un repère orthonormé appelé "repère de Monge".

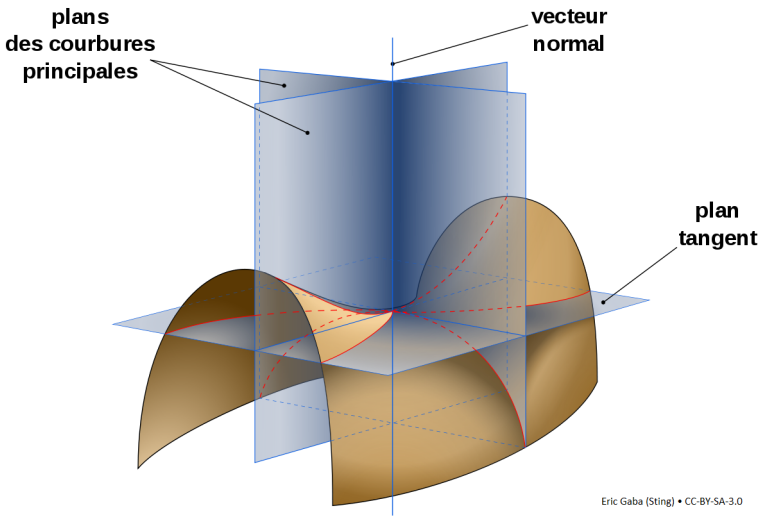


FIGURE 11 – Vue des plans définissant les courbures principales d'une surface en un point

Il existe cependant des points de la surface pour lesquels la courbure maximale et la courbure minimale sont égales. Cela signifie que toutes les courbures dans toutes les directions sont égales en ces points, et il est impossible d'y définir un repère de Monge. le point est alors appelé un ombilic.

Les lignes de courbure, qui nous intéressent pour notre application, sont des courbes appartenant à la surface qui sont tangentes aux directions principales en tous points. Il en existe une infinité et chaque point non ombilic est traversé par deux lignes de courbure, l'une tangente à la la direction principale maximale et l'autre tangente à la direction principale minimale, comme le montre la figure 12 ci-dessous.

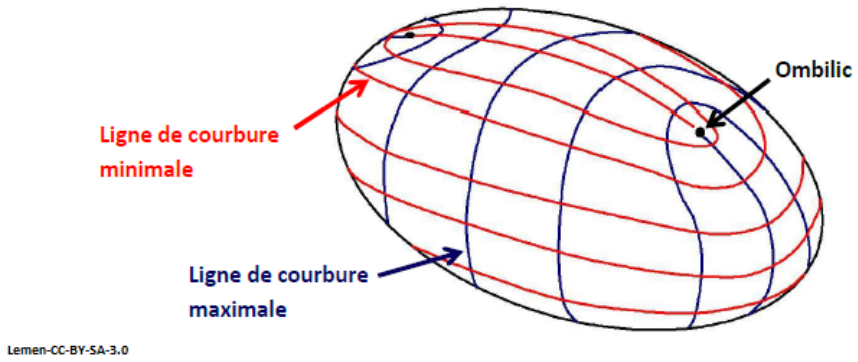


FIGURE 12 – Lignes de courbure et Ombilics

Les lignes de courbure extrémales (ou ridges) que l'on souhaite exploiter dans notre application

sont donc les lignes le long desquels les courbures principales sont extrémales localement (c'est à dire maximales ou minimales).

5.2.2 Implémentation

Comme nous l'avons déjà évoqué au chapitre précédent, le calcul des ridges va être effectué via une dll écrite en C++. A cette fin, nous allons utiliser la librairie CGAL qui implémente un algorithme et des structures de données permettant le calcul d'une approximation des ridges et des ombilics sur une surface discrétisée sous forme de maillage triangulaire. On le trouve dans le package *"Approximation of Ridges and Umbilics on Triangulated Surface Meshes"* [6]. Cet algorithme s'appuie sur les travaux de Marc Pouget et Frédéric Cazals dans [7] et [8] et de Yutaka Ohtake, Alexander Belyaev et Hans-Peter Seidel dans [9].

La documentation du package fournit un exemple pour calculer les ridges et les ombilics sur un maillage décrit par un fichier ".off". Mon travail d'implémentation en a été facilité et accéléré. la principale difficulté a été l'identification des informations à passer en paramètre et à récupérer pour notre application.

La particularité de cet algorithme est qu'il est très sensible à la qualité du maillage. Un maillage convenable devra avoir une densité suffisamment importante pour que l'approximation des ridges soit correct. Cependant, un filtrage à posteriori peut être opéré afin de réduire le bruit qui aurait été généré par le calcul des ridges sur un maillage de qualité non optimale. En effet il est possible d'effectuer un seuillage sur les valeurs de force (strength) et de finesse (sharpness) associées à une ridge calculée.

la structure de données qui permet de représenter et de stocker les ridges calculées dans le package CGAL est la classe *Ridge_Line*. Cette dernière stocke pour chaque ridge une valeur de *strength* et *sharpness* sur laquelle il sera possible d'effectuer un seuillage. Les coordonnées des ridges en elles-mêmes sont codées dans la structure sous forme d'une liste d'objets de type *Ridge_halfhedge*. Lequel est en fait un appariement entre un *halfhedge_descriptor*, objet servant à décrire une demi-arête du maillage, et les coordonnées barycentriques du point ou la ridge Calculée intersecte cette demi-arête. on peut donc récupérer les coordonnées de la ridge en parcourant toutes les *Ridge_halfhedge* et en récupérant les coordonnées des points d'intersections.

En résumé on va devoir passer les données suivantes à la dll :

— Le maillage que l'on souhaite analyser. Il sera envoyé sous la forme décrite au début de ce chapitre par la figure 9.

— Les deux valeurs de seuil pour réaliser le filtrage des ridges.

Et cette dernière renverra les coordonnées des points formant la ridge. A cette fin, on procédera presque de la même façon que pour le transfert du maillage en renvoyant un tableau de flottant de la forme :

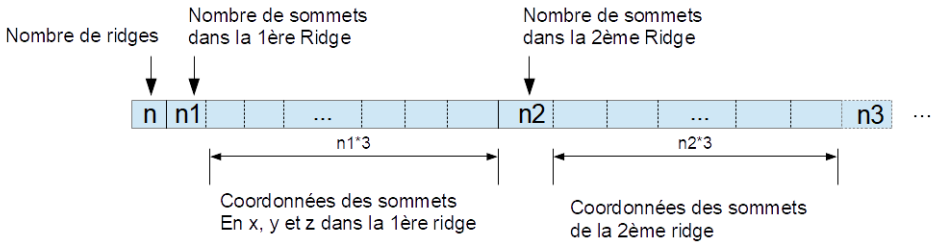


FIGURE 13 – Trame de retour pour le calcul des ridges

5.3 Complétion des zones détériorées

5.3.1 Triangulation, raffinement et lissage

C'est la première partie de la complétion de trous, la démarche illustrée à la figure 3 issue des travaux de P. Liepa décrits dans [1].

La première étape de la complétion des parties endommagées du maillage consiste à trianguler grossièrement la zone. Le principe est de trianguler le trou sans ajouter de nouveaux sommets, en cherchant à minimiser le plus important des angles dièdres (angle entre deux facettes), et à minimiser la surface du patch créé.

La seconde étape consiste à raffiner le maillage du patch créé. L'algorithme va effectuer un certain nombre d'opérations tels que des divisions, des fusions et/ou des basculements d'arêtes, ainsi qu'un lissage utilisant le Laplacien afin d'obtenir un maillage plus uniforme, à la densité proche de celle du maillage avoisinant, et s'approchant au mieux d'une triangulation de Delaunay.

Enfin, la troisième étape consiste à lisser le patch, en prenant en compte les contraintes définies par la configuration des bordures de la zone endommagée. Plus précisément, on va assurer la continuité des tangentes aux sommets du patch. le principe de l'algorithme qui réalise cette opération est de minimiser un système linéaire bi-Laplacien (voir [3]).

Lors du premier stage, une première implémentation de ces algorithmes avait été réalisée sous la plateforme Mepp. Nous avons pu récupérer les codes sources et les adapter pour créer une Dll, cependant pour certaines configurations de maillages cette implémentation posait des problèmes et l'opération sur le maillage n'aboutissait pas. Nous avons tenté de trouver d'où provenait le problème, mais parallèlement il est apparu que depuis l'époque du premier stage, ces trois algorithmes avaient été implémentés dans CGAL, au sein du package *Polygon Mesh Processing* [10]. Nous avons donc utilisé les fonctions de ce package pour créer notre dll. Nous reviendrons plus en détail sur la conception de cette dll dans la section 5.3.3.

5.3.2 Déformation du patch

La dernière étape de notre chaîne de traitement consiste à déformer notre maillage et plus précisément le patch construit afin de lui faire respecter une autre contrainte provenant non pas de la forme initiale du maillage mais d'une contrainte extérieure obtenue grâce à une manipulation des ridges sur laquelle nous reviendrons plus en détail par la suite. La solution retenue pour réaliser cette déformation va associer des contraintes de position à certains sommets du maillage. En dehors de la déformation en elle même dont nous allons parler maintenant, il est important de retenir que cela pose également deux problèmes : comment définir les contraintes de positions, et nous verrons cela lorsque nous aborderons l'exploitation des ridges, mais également comment allons-nous apparier ces contraintes à des sommets du maillage. Mais pour l'instant, nous allons nous pencher sur la déformation. Le principe repose également sur la résolution d'un système linéaire afin de minimiser une énergie de déformation. L'algorithme a également été implémenté assez récemment sur CGAL dans le package *Triangulated Surface Mesh Deformation* [11].

Le système de base pour parvenir à la déformation d'un maillage surfacique triangulaire est constitué des éléments suivants :

- un maillage surfacique triangulaire ;
- un ensemble de sommets de ce maillage constituant la zone à déformer (appelée ROI pour region-of-interest) ;
- un sous-ensemble de sommets de la ROI formant les sommets de contrôle. Ce sont les sommets que l'utilisateur souhaite soumettre à une nouvelle contrainte de position ;
- et enfin un ensemble de positions cibles pour les sommets de contrôle du maillage.

La figure 14 illustre ce système.

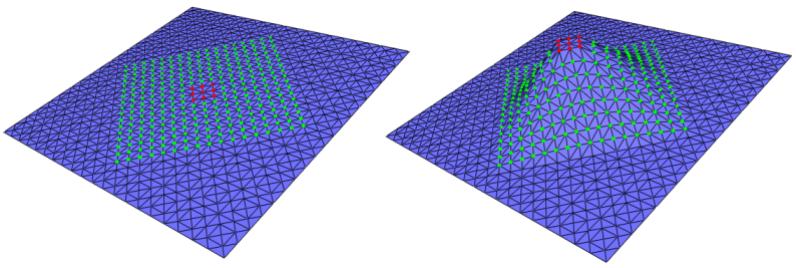


FIGURE 14 – Déformation d'un maillage surfacique triangulaire : en vert la ROI, en rouge les sommets de contrôle. Sur l'image de droite ces sommets sont déplacés à une position cible et la ROI est déformée en conséquence. Image issue de [11].

La technique de déformation utilisée par ce package est basée sur l'utilisation d'une représentation différentielle du maillage, et plus précisément sur l'utilisation d'une représentation Laplacienne. La représentation Laplacienne est une manière de décrire le maillage qui prend en compte la configuration du voisinage de chaque sommet. A chaque sommet Vi est associé un vecteur 3D défini de la façon suivante :

$$L(Vi) = \sum_{Vj \epsilon N(Vi)} Wij(Vi - Vj)$$

où :
— $N(Vi)$ désigne l'ensemble des sommets voisins de Vi
— Wij un poids pour l'arête $ViVj$

Si l'on choisit par exemple un poids uniforme pour tous les Wij, la représentation Laplacienne de Vi correspondra au vecteur entre ce sommet et le barycentre de ses voisins (illustré par l'annexe B).

Cependant le choix le plus populaire et celui qui est proposé par défaut dans le package est le poids cotangent. Pour une arête donnée, son poids cotangent est la moyenne des valeurs des cotangentes des angles opposés à cette arête. Il est connu pour produire des résultats satisfaisants dans la plupart des configurations aussi nous n'avons pas essayé d'autres valeurs pour le poids, mais on note que les fonctions du package offrent la possibilité de le faire.

Cette représentation Laplacienne du maillage peut s'écrire sous forme matricielle de la manière suivante :

En considérant un maillage composé de n sommets, sa représentation Laplacienne sera une matrice Δ de taille $n * 3$ tel que :

$$LV = \Delta \tag{1}$$

où :
— L est une matrice $n * n$, appelée matrice Laplacienne dont les éléments $m_{ij}, i, j \epsilon 1..n$ sont définis de la manière suivante :
  — $m_{ii} = \sum_{Vj \epsilon N(Vi)} Wij$,
  — $m_{ij} = -Wij$ si $Vj \epsilon N(Vi)$,
  — 0 sinon.
— V est une matrice $n * 3$ des coordonnées cartésiennes des sommets du maillage.

Cette base étant posée, le principe général de la déformation Laplacienne est de déformer des points du maillage vers une position cible tout en préservant la représentation Laplacienne du maillage. De manière intuitive, on comprend que l'on cherche à déformer le maillage tout en conservant la configuration du voisinage de chaque sommet.

A cette fin, on va établir, afin de modéliser le système de déformation et ses contraintes, l'équation ci-dessous sous forme matricielle ;

En considérant un système de déformation de maillage constitué d'une ROI de n sommets dont k sommets de contrôle, on a le système linéaire suivant :

$$\begin{bmatrix} L_f \\ 0 \quad I_c \end{bmatrix} V = \begin{bmatrix} \Delta_f \\ V_c \end{bmatrix} \tag{2}$$

où :
— V est une matrice $n * 3$ correspondant aux coordonnées des sommets de la ROI après déformation, soit les inconnues du système. Elle est construite de sorte à ce que les k dernières lignes correspondent aux sommets de contrôle.
— L_f correspond à la matrice Laplacienne des sommets de la ROI non contraints (ceux qui ne font pas parties des sommets de contrôle). C'est donc une matrice de taille $(n - k) * n$, identique à la matrice L de l'équation 1 à ceci près que l'on a supprimé les lignes correspondant aux sommets de contrôle.
— I_c est la matrice identité de taille $k * k$.
— Δ_f correspond à la représentation Laplacienne des sommets non contraints de la ROI. C'est donc une matrice $(n - k) * 3$, équivalente à la matrice Δ dans l'équation 1 à laquelle on a supprimé les lignes correspondant aux sommets de contrôle.
— V_c est une matrice $k * 3$ contenant les coordonnées cartésiennes des positions cibles des sommets de contrôle.

Ainsi, résoudre ce système permet de conserver la représentation Laplacienne du maillage tout en satisfaisant les contraintes de déformation. Cependant un système uniquement basé sur cette

approche pour la déformation ne sera pas idéal car il en résultera des transitions trop brutales dans le voisinage des points de contrôle. C'est pourquoi l'algorithme implémenté par CGAL inclut dans le système une fonction d'énergie. Il y en a en fait trois disponibles dans le package selon que l'on souhaite une déformation de type *As-Rigid-As Possible* [12], *Spokes and Rims* [13] qui est celle par défaut et que nous utiliserons, ou *Smoothed Rotation Enhanced As-Rigid-As Possible* [14]. Mais au final cela revient à la résolution d'un système linéaire équivalent à celui de l'équation 2.

5.3.3 Conception du plugin

Alors que le premier plugin décrit au début de ce chapitre (5.2.2) servait uniquement à réaliser l'opération de calcul des ridges sur le maillage passé en paramètre, celui-ci va réunir un nombre d'outils et de fonctions plus importants pour manipuler le maillage, et échanger d'autres types de données. En conséquence, il est apparu nécessaire d'organiser les codes C++ générant la dll utilisée dans Unity de manière plus rigoureuse. A cette fin, nos avons réalisé un projet exploitant le logiciel Cmake. Cmake est un "moteur de production" multi-plateforme, qui va permettre de compiler un projet en C/C++ quelque soit le système d'exploitation, le compilateur ou même l'environnement de développement utilisé. En effet, Cmake ne produit pas le logiciel final, mais va générer grâce au instructions écrites dans un fichier "CMakeLists.txt" les fichiers de constructions standard propre à chaque plateforme, par exemple un makefile sous Linux, ou un projet Visual Studio sous Windows. On obtient un projet qui n'est plus dépendant de la plateforme utilisée.

Une illustration de la structure du projet est donnée en annexe C. A chaque niveau de l'arborescence les fichiers "CmakeLists.txt" donnent les instructions nécessaires à la génération des fichiers de construction par CMake.

- Les fichiers des répertoires "src/" et "include/" correspondent aux fonctions principales de la dll.
  - En particulier les fichiers *Fill_Holes.cpp/.h* servent à décrire la classe principale du programme, *Fill_Holes*. C'est cette classe qui manipule les données relatives au maillage et au patch que l'on est en train de reconstruire. Les fonctions de cette classe font appel aux fonctions de la librairie CGAL pour réaliser les opérations liées à la complétion du trou décrites dans les sections précédentes.
  - Les fichiers *Utils.cpp/.h* implémentent les fonctions outils permettant de convertir les données arrivant sous forme de trames du type de celle décrite dans la figure 9 en données exploitable Côté C++ par CGAL, ainsi que les fonctions permettant de faire la passerelle entre la Classe *Fill_Holes* et les fonctions des fichiers du répertoire "apps".
  - enfin le fichier *polyhedron_builder.h* fournit une classe permettant de construire un objet de la classe *polyhedron_3*, qui est, on le rappelle, la structure de donnée permettant de décrire les maillages surfaciques 3D dans CGAL, à partir de données sous forme de listes de sommets et de facettes. Et le fichier *items.h* décrit une classe qui est un modificateur permettant de personnaliser la structure de données en demi-arête de la classe *Polyhedron_3*. On peut ainsi modifier les classes de base décrivant les sommets, les arêtes et les facettes afin d'y ajouter des données supplémentaires. Nous verrons par la suite que nous nous en servons pour ajouter des indices aux sommets du maillage, nécessaires pour exécuter un algorithme d'appariement entre les positions cibles de la déformation et les points de la ROI.
- Les fichiers du répertoire "apps/" sont répartis en deux répertoires : "FillHoleExample/" qui contient les codes permettant de tester le programme grâce à un exécutable, et "FillHolePlugin/" qui contient les fichiers *export.h* et *interface.cpp* permettant de créer l'interface pour la dll générée.

En résumé, on peut voir la dll comme une boîte à outils donnant accès aux fonctions dont voici les prototypes :

```
double* compute_triangulate_refine(double* stream)
```

Cette fonction va réaliser la création d'un patch par triangulation d'un trou du maillage détecté automatique et le raffinement de ce patch. Elle prend en paramètre un pointeur vers un tableau de double qui décrit le maillage de la manière décrite dans la figure 9. Elle renvoie un pointeur vers un tableau de 'double' qui décrit le maillage traité sous forme de trame et contient également les indices des sommets du maillage qui correspondent au patch nouvellement formé.

```
double* compute_fair(double* stream, int* index, int lgth)
```

Cette fonction va réaliser un lissage du patch, en assurant la continuité des tangentes.
Ses paramètres :
— stream est la trame décrivant le maillage ;
— index est un tableau des indices des sommets du maillage formant le patch que l'on souhaite lisser ;
— lgth est la longueur du tableau index.
Comme la première fonction, elle renvoie le maillage modifié sous forme de trame.

```
double* compute_deform(double* stream, int* index,
        int lgth, double* pos, int posLgth)
```

Cette fonction va réaliser la déformation du patch sous contraintes extérieures.
ses paramètres :
— stream est la trame décrivant le maillage ;
— index est un tableau des indices des sommets du maillage formant le patch que l'on souhaite déformer. Ce patch constituera la ROI du système de déformation ;
— lgth est la longueur du tableau index.
— pos est le tableau contenant les positions cibles de la déformation.
— posLgth est la longueur de pos.
Si l'on se rappelle des explications données dans la section 5.3.2, il manque un élément important pour la construction du système de déformation ; il s'agit des points de la ROI directement soumis aux contraintes de position, les points de contrôle. Nous disposons de positions cibles mais nous ne savons pas comment les appliquer au maillage. Dans cette fonction, nous avons tenté d'apparier automatiquement les positions cibles à des sommets du maillage. À cette fin, nous avons implémenté l'algorithme suivant :

Data: positions cible $\{p_0, ..., p_m\}$, sommets du patch $\{v_0, ..., v_n\}$
Result: $p_i.candidat$, l'indice du sommet du patch apparié au sommet $p_i \forall i \in \{0, ..., m\}$

for $i \leftarrow 0$ **to** n **do**
  $v_i.candidat \leftarrow j$;
  // j tel que $min_{0 \leq j \leq m} ||v_i - p_j||$ est atteint
end
for $j \leftarrow 0$ **to** m **do**
  $p_j.candidat \leftarrow i$;
  // i tel que $min_{v_i.candidat=j} ||v_i - p_j||$ est atteint
end

L'idée de cet algorithme est d'abord d'apparier chaque sommet du patch à une position cible qui lui est la plus proche parmi toutes les positions cibles, puis de parcourir toutes les positions cibles en sélectionnant pour chacune, parmi les sommets du patch qui ont indiqué que cette position cible était leur plus proche, celui qui est le plus proche.

Si cette approche a été utilisée dans un premier temps plutôt que d'apparier directement chaque position cible au sommet du maillage le plus proche, c'est parce que, comme nous le verrons dans la section suivante (5.4.3), certaines positions cibles obtenues côté Unity3D et passées en paramètre à la fonction ne sont pas pertinentes. Or cela permettait automatiquement de ne pas prendre en compte ces positions non pertinentes, car elles ne sont jamais sélectionnées lors de la première phase de l'algorithme, étant trop loin de tout sommet du patch. Cependant pour cette même raison des sommets significatifs sont également ignorés et les résultats ne sont pas conformes aux attentes.

Nous avons donc choisi d'envisager une approche différente.

```
compute_deform2(double* stream, int* index,
        int lgth, double* pos, int posLgth, int* control, int controlLgth)
```

Cette fonction réalise les mêmes opérations que la précédente, sauf que les sommets du patch et les positions cibles ne sont pas appariés automatiquement. Ici les sommets de contrôle sont passés en paramètre à la fonction. Nous verrons dans la section 5.4.3 comment nous les avons obtenus.

Maintenant que nous avons un aperçu complet de l'implémentation et du fonctionnement des fonctionnalités principales liées à la complétion des maillages endommagés, nous allons voir comment les utiliser du côté du projet Unity3D.

5.4 Visualisation et interaction

Pour rappel, nous avons choisi d'implémenter dans un premier temps une solution qui exploite les symétries d'un modèle afin de compléter ses parties manquantes, mais d'appliquer cette symétrie non pas directement sur le maillage mais sur un modèle sémantique simplifié de notre maillage, obtenu par le calcul des ridges qui nous donne les formes caractéristiques de l'objet (voir figure 4 pour un rappel de la procédure envisagée).

5.4.1 Cas d'utilisation

Afin de rendre compte des fonctionnalités qui seront présentes dans notre application, vous trouverez ci-dessous un diagramme Use Case UML (figure 15).

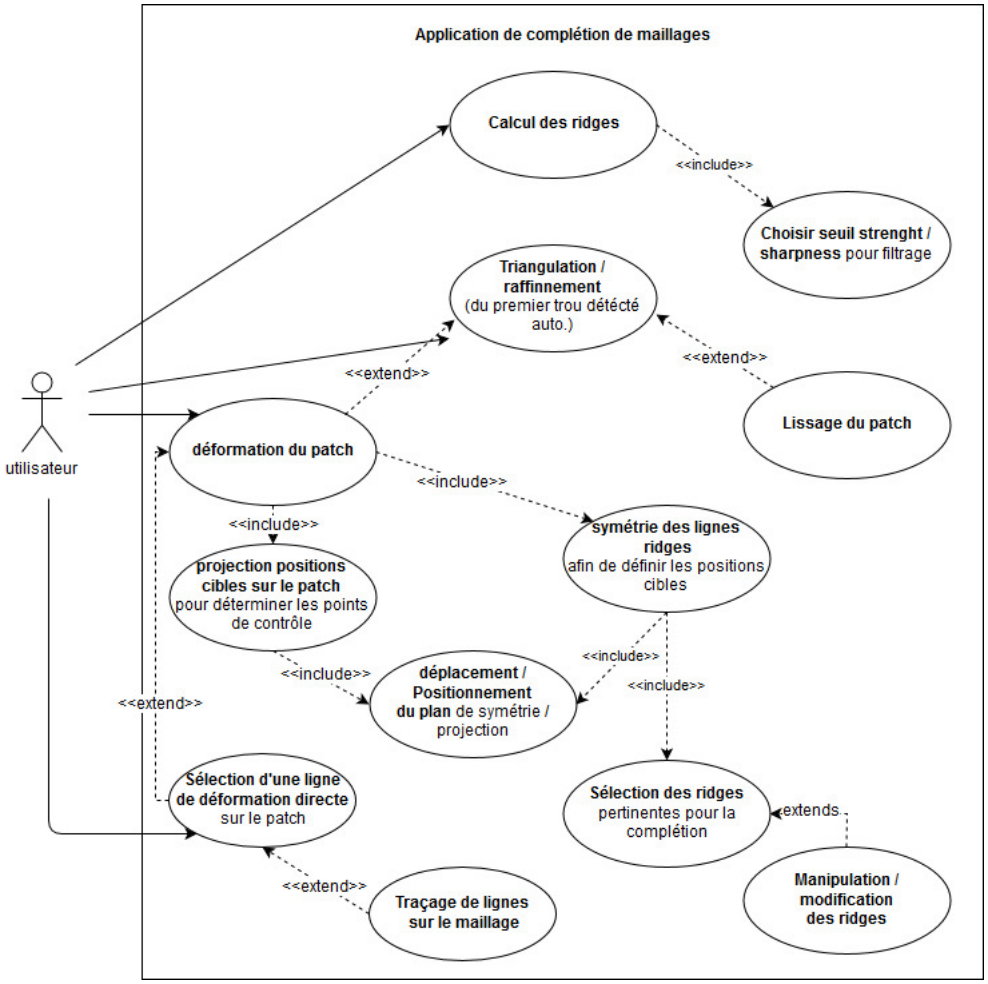


FIGURE 15 – Diagramme Use Case de l'application

Pour résumer simplement ce diagramme, on peut dire que l'utilisateur va interagir à tous les niveaux de la complétion du maillage. D'abord, au moment du calcul des ridges en spécifiant le filtrage qu'il souhaite appliquer. Les opérations de triangulation, raffinement, et lissage du patch quant à elles sont réalisées automatiquement et ne nécessitent pas d'intervention particulière de l'utilisateur, sinon pour les déclencher. Enfin la déformation est pilotée par l'utilisateur de deux manières possibles : soit en exploitant les symétries sur les ridges en vue de définir des contraintes de position pour la déformation en réalisant un certains nombre d'opérations sur lesquels nous reviendrons plus en détail dans la section 5.4.3, soit en sélectionnant et déformant directement une

ligne sur la surface, dite ligne d'édition, qui peut être ou bien une ligne tracée par l'utilisateur (voir section 5.4.5), ou bien une ligne de symétrie (c'est-à-dire une ligne de positions cibles) après déformation, et que l'on souhaite ajuster.

5.4.2 Gestion du bras haptique et du casque de réalité virtuelle

La majorité des interactions de l'utilisateur sont pilotées par le bras haptique. On va notamment s'en servir pour :

— sélectionner les ridges/lignes à la surface de l'objet ;
— déplacer le plan de symétrie/projection ;
— tracer des lignes sur la surface ;
— éditer/modifier les lignes sélectionnées.

Comme nous l'avions déjà évoqué dans le chapitre 4 à la section 4.2.1, l'utilisation du bras haptique dans Unity s'est faite de manière assez directe grâce au package "Haptic Plugin for Geomagic OpenHaptics 3.3". Pour utiliser ce package, il faut inclure un GameObject dans notre scène dont voici la description :

GameObject Dummy

Il contient un ensemble de scripts permettant de paramétrer le fonctionnement du retour haptique dans la scène. Un ensemble de Scripts C# est utilisé pour faire l'interface avec un Plugin écrit en C++ et qui utilise l'API OpenHapticToolkit pour implémenter les fonctionnalités du retour de force dans l'environnement Unity3D.

Composants principaux :

— Script -> GenericFunctionClass : Ensemble des fonctions permettant d'initialiser et de gérer l'environnement haptique dans la scène, ses propriétés mécaniques et physiques.
— Scripts -> ConstantForceEffect, ViscosityEffect, SpringEffect, FrictionEffect : Ces scripts permettent de stocker un certain nombre de propriétés relatives aux propriétés physiques et mécaniques du retour haptique.
— Script -> SimpleShapeManipulationDynamic : script qui lance l'exécution de l'environnement Haptic dans la scène Unity. C'est le seul script que nous avons légèrement modifié par rapport à l'original, afin qu'il puisse recalculer l'environnement haptique lorsque de nouveaux objets apparaissent dans la scène.

Il fallait également un objet permettant de faire le rendu en temps réel du curseur dans la scène virtuelle. Il s'agit du **GameObject Curseur**. Pour qu'une instruction soit envoyée au bras et qu'un retour de force soit calculé en cas de collision avec un GameObject dans la scène virtuelle, il faut que ce GameObject possède le composant "Haptic Properties", un script permettant de stocker un ensemble des propriétés relatives au retour produit au contact de la surface du maillage de l'objet.

Nous utilisons également un casque de réalité virtuelle afin d'augmenter l'immersion de l'utilisateur et de visualiser en 3D l'objet à compléter. Son utilisation dans la scène Unity est également assez simple grâce au package "SteamVR Plugin". Ce package fournit deux prefabs (des GameObjects préconstuits), **steamVR** qui permet de gérer différents paramètres globaux pour la réalité virtuelle, et notamment l'espace tracké par le système de caméra, et **CameraRig** qui représente la caméra donnant la vue du casque de réalité virtuelle et auquel sont attachés les composants trackés comme les contrôleurs. Il suffit de placer ces deux gameObjects dans la scène pour utiliser le casque de manière simple.

5.4.3 Exploitation des ridges et des symetries

Dans cette section nous allons parler de la manière dont nous exploitons les ridges pour construire nos contraintes de position pour la déformation. Les explications que nous allons donner se rapporteront donc aux actions du diagramme Use Case de la figure 15 suivante : sélection des ridges, déplacement et positionnement du plan, calcul de la symétrie, et projections des lignes de symétrie sur le patch.

L'idée est la suivante : après avoir reconstruit le patch sur la surface trouée, on souhaite le déformer pour lui faire respecter une certaine forme, en se basant sur la forme d'une partie symétrique intacte. A cette fin nous allons procéder de la manière suivante :

1. On sélectionne les ridges sur la partie intacte dont on souhaite réaliser la symétrie par rapport à un plan.

2. on va créer et manipuler ce plan à notre convenance pour le positionner comme on le souhaite puis on calcule les lignes symétriques. On peut réajuster leur position en temps réel en manipulant le plan.
3. Ces lignes symétriques vont constituer les positions cibles de notre déformation. On réalise une projection sur la surface du patch dont la direction est orthogonale au plan manipulable, afin d'apparier les positions cibles avec les sommets du patch qui feront office de sommets de contrôle.
4. On passe enfin tout ce système en paramètre à la fonction de la dll qui réalise la déformation.

Afin d'illustrer la conception de ces différentes fonctionnalités sous Unity 3D, vous trouverez ci-dessous un diagramme proche d'un diagramme de classes UML (Figure 16), adapté cependant à la structure des projets Unity où tous les objets d'une scène sont représentés par des GameObjects constitués de composants spécifiant leur comportement. Ces objets peuvent être liés entre eux par les scripts, mais également par des liens de parenté entre GameObjects. L'utilisation de base du lien de parenté vise à attribuer le même système de coordonnées aux objets fils d'un objet parent dans la scène, mais on peut aussi passer facilement de l'un à l'autre dans les scripts ce qui peut être pratique pour récupérer des données.

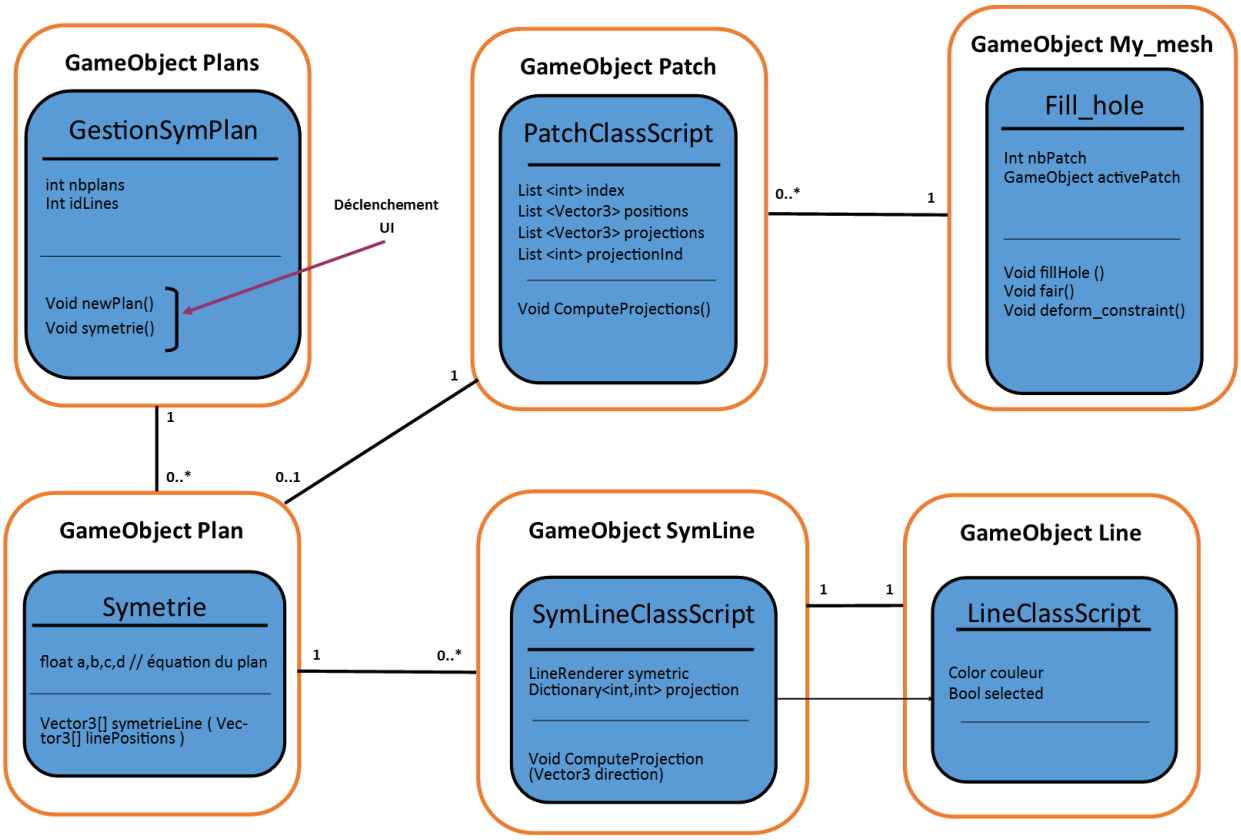


FIGURE 16 – Diagramme représentant les GameObjects et les scripts principaux intervenant dans l'opération permettant de réaliser la complétion de trou en exploitant la symétrie des ridges

1. **GameObject My mesh** : C'est le GameObject principal de l'application, il correspond au maillage sur lequel on travaille et que l'on souhaite réparer.

   Composants principaux :
   — Mesh Filter : composant décrivant le maillage de l'objet.
   — Mesh Renderer : propriétés pour le rendu du maillage.
   — Mesh Collider : permet au moteur physique de Unity3D de détecter la collision avec l'objet à partir de son maillage.
   On retrouvera des composants similaires à ces trois premiers dans tous les objets physiques de la scène.
   — Script -> Compute_Ridges : ce script appelle la Dll C++/CGAL et lui fournit les paramètres nécessaires pour le calcul des lignes de courbure extrémales.
   — Script -> Fill_Hole : permet la triangulation des trous et le raffinement du maillage obtenu, le lissage du patch obtenu et sa déformation suivant nos contraintes. En bref tous les outils relatifs à la complétion du trou suivant les objectifs que l'on s'est fixé.

2. **GameObject Plans** : Ce gameObject parent est un conteneur pour les GameObject enfants correspondant à des plans de symétrie qui seront créés dynamiquement à la demande de l'utilisateur à l'exécution.

   Composants principaux :
   — script -> GestionSymPlan : Ce script permet de lancer la création des plans de symétrie et les symétries par rapport à un plan sélectionné par l'utilisateur (en l'état actuel le plan actif est le dernier plan créé mais on envisagera une sélection manuelle)

3. **GameObject Plan«i»** cet objet est un plan de symétrie crée à la demande de l'utilisateur. Ce dernier va pouvoir le déplacer puis le positionner pour appliquer la symétrie qu'il désire.

   Composants principaux :
   — Script -> Symétrie : calcule les symétriques des lignes sélectionnées dans la scène par rapport au plan de l'objet courant.

4. **GameObject Patchs** : Ce gameObject parent est un conteneur pour les GameObject enfants correspondant à des patchs qui seront créés lorsqu'un trou sera complété par l'utilisateur.

5. **GameObject Patchs«i»** : Cet objet correspond à un patch créé à l'exécution. Il n'est pas matérialisé dans la scène.

   Composants principaux :
   — Script -> PatchClassScript : permet de stocker les indices des sommets du maillage correspondant au patch. Permet également de réaliser l'opération de projection pour déterminer les sommets du patch qui correspondront à des point de contrôle lors de la déformation contrainte de ce dernier.

6. **GameObject line_ «i»** : Correspond à une poly-ligne créée dynamiquement à l'exécution de la fonction ComputeRidges.ridges rattachée au gameObject *My_ mesh*. Les GameObjects *line* de notre scène sont des éléments fils du GameObject *My_ mesh*. Ainsi, ils subissent les mêmes transformations que ce dernier.

   Composants principaux :
   — LineRenderer : permet d'afficher la ligne dans la scène Unity et de stocker ses positions et ses paramètres de rendu ;
   — Script -> LineClassScript : permet de stocker des informations sur la ligne, sa couleur, et son état (sélectionnée ou non) ;
   — Script -> LineCollider : permet de créer un maillage autour des lignes pour initialiser le Mesh Collider de l'objet courant, qui permettra de détecter la collision entre les lignes et le curseur pour l'interaction.

7. **GameObject SymLine«i»** : Correspond à une poly-ligne créée dynamiquement par symétrie par rapport à un plan d'une poly-ligne correspondant à une ridge (GameObject *line_ «i»*).

   Composants principaux :
   — Script -> SymLineClassScript : Classe héritée de la classe LineClassScript. permet en plus de stocker le LineRender de la ligne dont elle est le symétrique afin de pouvoir recalculer la symétrie en temps réel si l'on déplace le plan. Une autre fonction de ce script permet également le calcul de la projection de la ligne sur la surface suivant une

direction et de sauvegarder les paires "points de contrôle/destinations cibles" ainsi créées dans un *Dictionary* ;

5.4.4 Edition des ridges

Comme nous l'avons signalé à la section 5.2.2, la qualité des ridges obtenues par l'algorithme est très dépendante de la qualité du maillage, et notamment de sa densité. Aussi, nous avons décidé de tenter de mettre en place un outil permettant de manipuler et modifier les ridges, afin de les ajuster à notre convenance. Pour y parvenir, nous allons procéder de la manière suivante :

- Les lignes ridges sont ré-échantillonnées de manière régulière après leur calcul.
- En passant en mode édition, on peut agir à l'aide du bras haptique sur les points de contrôle d'une courbe sélectionnée. Ces points de contrôle sont en l'état actuel les points de la courbe entre lesquels cette dernière est interpolée de façon linéaire, cependant on pourrait envisager d'utiliser des splines par la suite si cela s'avérait nécessaire. Le logiciel a été conçu avec cette idée en différenciant les points de la courbe des points de contrôle, qui sont stockés grâce au script *LineClassScript* de chaque ligne.
- Lorsqu'un point de contrôle est déplacé, ce déplacement est diffusé aux sommets voisins suivant la fonction sinusoïdale suivante :

$$\frac{1}{2} + \frac{1}{2}\cos^{\alpha}(\frac{d}{W}\pi)$$

où :

- d est le déplacement du point de contrôle ;
- W est le nombre de points de chaque côté du point de contrôle qui subissent l'influence du déplacement ;

α, l'exposant du cosinus, va avoir une influence sur la forme de la courbe déformée, tel qu'illustré dans l'annexe D.

Ces fonctionnalités sont codées dans le script *LineEdition*, composant du **Gameobject Curseur**.

5.4.5 Traçage et déformation directe du maillage

Enfin, la dernière fonctionnalité que nous avons mis à disposition de l'utilisateur est le traçage direct de lignes sur la surface du maillage. Cela peut être soit des lignes qu'on trace sur le maillage à l'aide du bras haptique et auxquelles on appliquera la même procédure de symétrie pour définir des positions cibles de contrainte qu'aux ridges, soit des lignes que l'on trace directement sur le patch, dans le but de le déformer. En effet, on peut désigner une ligne positionnée sur le patch comme ligne d'édition, et la modification de cette ligne entraînera la modification du maillage. La technique de déformation utilisée ici est la même que celle décrite précédemment, la seule différence étant que les points choisis comme points de contrôle sont les points du maillage les plus proches de la courbe dans sa position initiale, et les positions cibles sont les positions de la courbe déplacée.

Un autre cas que celui de la ligne tracée à la main permet également la déformation directe : on désigne comme ligne d'édition une ligne obtenue par symétrie de ridge. Après une première déformation suivant la procédure standard, ces lignes se retrouvent à la surface du maillage reconstruit. On peut alors ajuster cette reconstruction en direct en modifiant cette ligne d'édition.

Cette déformation n'est pas tout à fait exécutée en temps réel. En effet, on modifie la courbe en temps réel mais le maillage n'est déformé que lorsque l'on relâche la courbe. La déformation de la surface dans notre application a un temps d'exécution trop important pour être réalisée en temps réel. Nous reviendrons sur ce point plus en détail dans la section 6.4. Cependant cette alternative permet de déformer le maillage de manière interactive.

Ces fonctionnalités sont implémentées dans les scripts *LineDrawing* et *LineEdition* du **GameObject Curseur**.

6 Résultats et analyse

6.1 Evaluation de l'utilisation des ridges

Comme nous l'avons expliqué, l'exploitation des ridges comme outil pour obtenir un modèle sémantique simplifié du maillage à compléter a été proposée lors du précédent stage car ces dernières permettent de rendre compte des formes caractéristiques à la fois d'objets arrondis et d'objets aux arêtes vives. Le principe avait été validé sur des formes simples, cependant le temps avait manqué pour faire une exploration plus approfondie du calcul des ridges sur des modèles plus complexes. On sait notamment que l'algorithme est sensible à la qualité du maillage fournit en paramètre. Sans prétendre faire une étude exhaustive, nous allons donner quelques résultats avec différents modèles.

1. L'ellipsoïde ci-dessous est représentée par un maillage comptant 9157 sommets et 18164 faces.

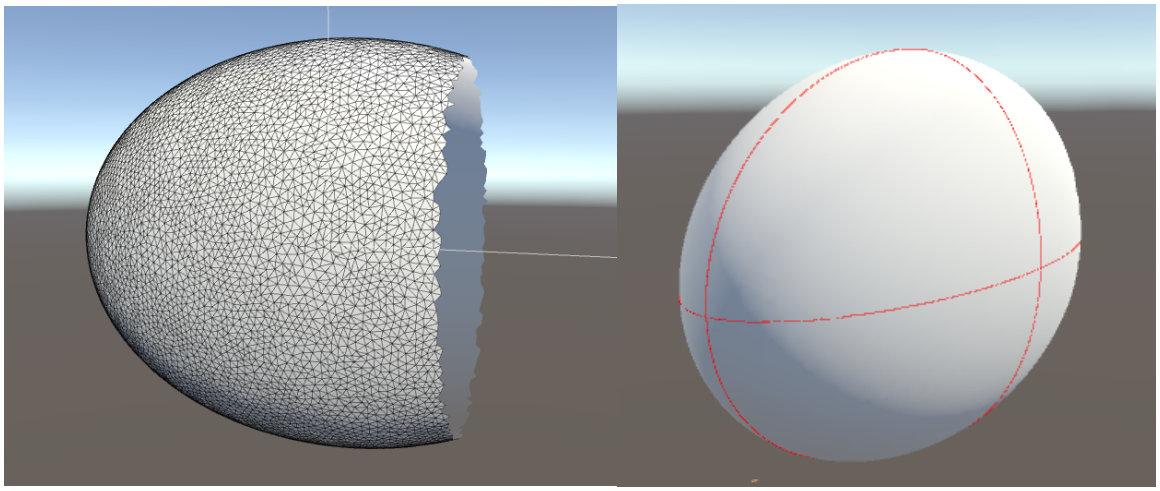


FIGURE 17 – Calcul des ridges sur le maillage d'une ellipsoïde endommagée

Les ridges obtenues ici sont nettes, elles seront facilement exploitables pour notre procédure. Il est par contre important de noter que cette opération ne se fait pas de manière instantanée : son temps d'exécution est sur ce modèle de l'ordre d'une dizaine de secondes. Ce n'est pas forcément gênant car cette opération est réalisée tout au début et n'a pas obligatoirement besoin d'être effectuée en temps interactif.

2. Le modèle du "Stanford Bunny" compte 16979 sommets et 33907 facettes. sa densité est comparable à celle de l'ellipsoïde, cependant sa géométrie est plus complexe.

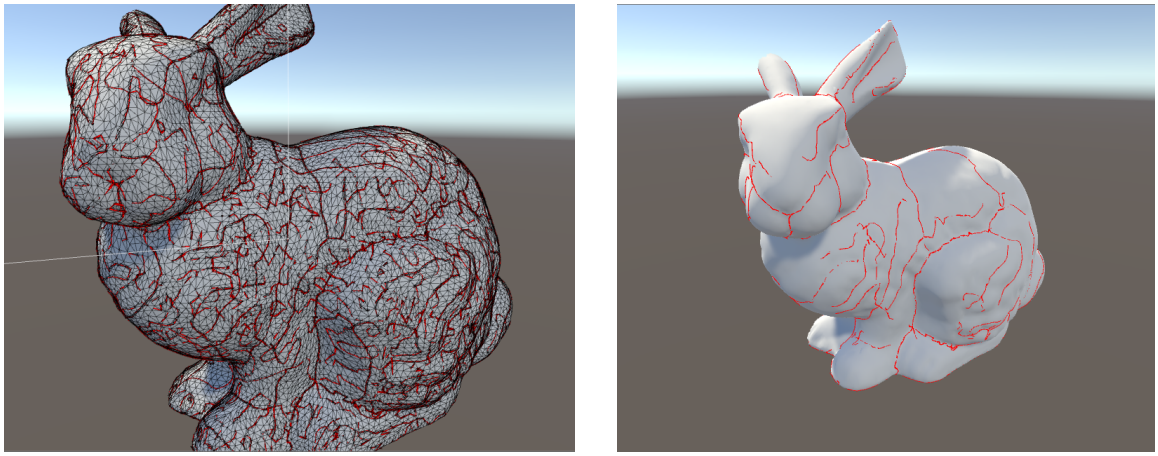


FIGURE 18 – Calcul des ridges sur le maillage du "Stanford Bunny". A gauche aucun filtrage n'a été opéré et à droite un filtrage de valeur 1 pour strength et 10000 pour sharpness

On constate que le résultat sur l'image de gauche est fortement bruité. Il y a plus de lignes que dans le premier exemple et le calcul en est d'autant plus long : l'opération prend une petite trentaine de secondes. Même si cela n'est pas critique, il pourrait tout de même être intéressant de réduire ce temps pour pouvoir gérer interactivement les paramètres de filtrage.

L'image de droite justement illustre un résultat après filtrage. Les ridges sont encore légèrement bruitées mais on peut voir apparaître des lignes intéressantes. Mais ce modèle simplifié est-il suffisant pour nos traitements ?

Cela dépend en fait des cas, de la position des parties manquantes. Sur ce modèle on peut trouver au moins un contre-exemple important, que l'on va illustrer maintenant à travers la figure ci dessous :

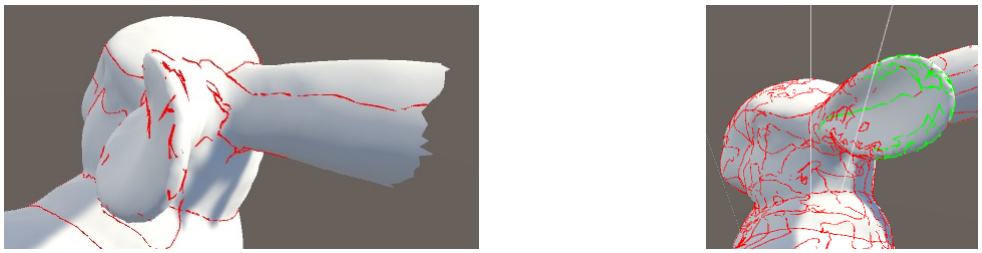


FIGURE 19 – Ridges sur la surface des oreilles du "Stanford Bunny"

Considérons qu'une des deux oreilles est endommagée et que l'on souhaite la reconstruire en exploitant la symétrie des ridges sur la deuxième oreille intacte. Le résultat filtré avec les paramètres précédents (image de gauche) ne permet pas d'extraire des ridges satisfaisantes pour rendre compte de la forme de l'oreille. Le problème est que même lorsque le résultat n'est pas du tout filtré (image de droite) on ne trouve pas, parmi le bruit, de lignes qui rendent compte convenablement de la forme de l'oreille. Sans remettre en cause fondamentalement l'utilisation des ridges comme support d'information dans notre chaîne de traitement, cela montre que dans certains cas elle n'est pas aisée. On peut envisager les éditer si elles ont de petits défauts mais lorsqu'elle sont aussi approximatives que celles de la figure 19 ci-dessus, cela n'est pas forcément possible ou pertinent. Ce qu'il faudrait donc c'est identifier de manière plus précise les cas qui produisent des résultats valides et ceux qui vont en produire de moins bons. Je n'ai pas eu le temps de plus approfondir la question durant le stage, mais la question de la validité des maillages passés en paramètres au calcul des ridges est discutée dans [8] et il serait pertinent de s'y intéresser de plus près.

6.2 Complétion de maillages endommagés

Nous allons illustrer la chaîne de traitement de notre application en prenant pour exemple le maillage de l'ellipsoïde endommagée utilisée dans la section précédente. Pour comparer, la figure 20 montre quels sont les résultats sans l'apport des ridges.

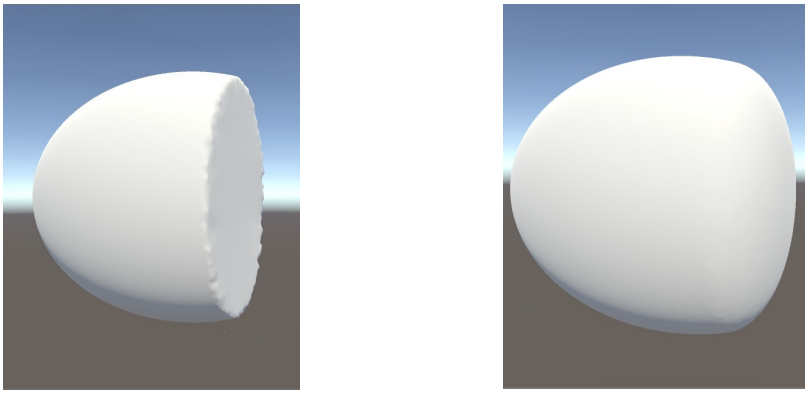


FIGURE 20 – résultats obtenus après triangulation, raffinement (image de gauche) puis lissage (image de droite) du patch

L'ordre des opérations décrit par la figure 21 est celui qui donne le meilleur résultat. On pourrait également refaire un raffinement du maillage déformé à la fin du traitement pour avoir un maillage

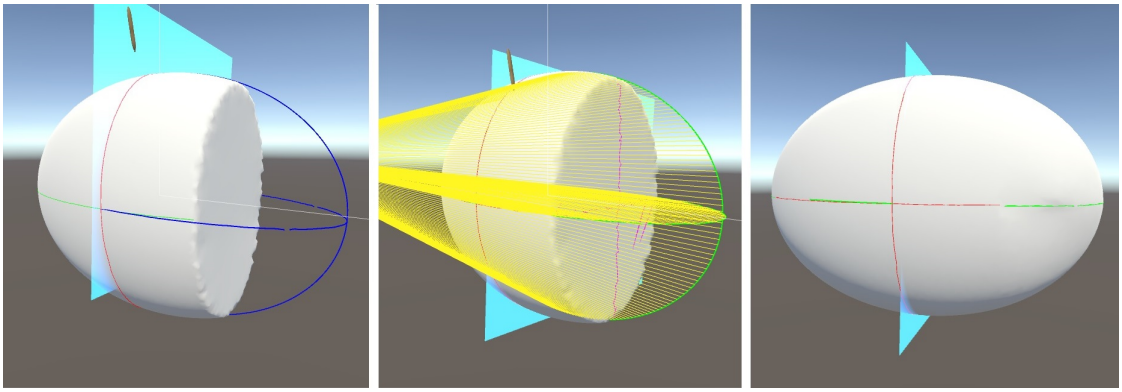


FIGURE 21 – Symétrie des ridges par rapport à un plan, projection dans la direction de la normale à ce plan, puis déformation avec position contraintes et lissage

plus homogène.

Les résultats sur ce genre de forme sont plutôt encourageants, l'utilisation des ridges comme contrainte de position fonctionne et on aboutit à une forme proche de celle que l'on souhaite instinctivement obtenir.

Si on s'intéresse maintenant au modèle du "Stanford Bunny", sur lequel nous souhaitons reconstruire l'oreille endommagée, le problème est un peu plus compliqué. Comme nous l'avons signalé dans la section précédente, les ridges sont difficilement exploitables. On peut cependant essayer de sélectionner au hasard quelques lignes sur l'oreille intacte tel qu'illustré à la figure 19. En réalisant notre procédure en se basant sur ces lignes, on obtient le résultat illustré à la figure 22.

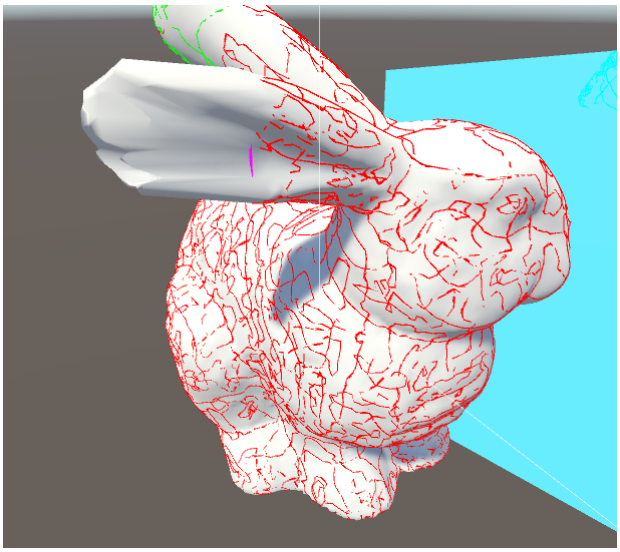
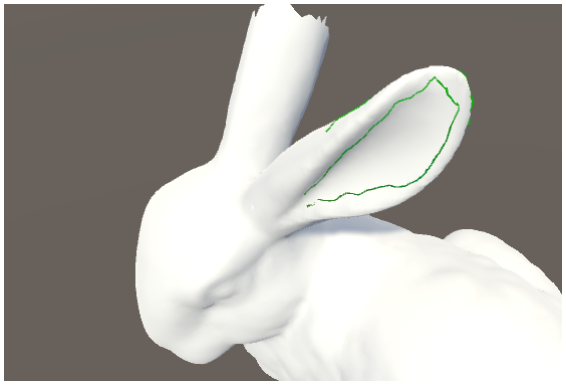


FIGURE 22 – résultat de la complétion sur l'oreille du "Stanford Bunny"

Même si le résultat présente quelques défauts, on retrouve bien la forme de l'oreille. Le principal inconvénient est que le résultat est un peu aléatoire ; en effet tout dépend de la manière dont va s'exécuter la projection. Les lignes définissant les contraintes de positions se chevauchent, s'entremêlent. Or dans l'opération de projection telle qu'elle est implémentée actuellement, si deux points sont projetés sur un même point du maillage, le deuxième ne sera pas pris en compte. Toute information perdue lors de la projection est perdue pour la reconstruction. Cependant le fait qu'un nombre important de lignes soit sélectionné et qu'elles définissent un nombre assez important de contraintes de positions va compenser ces petits défauts, même si de petits artefacts apparaissent, qu'on ne peut pas toujours supprimer avec la fonction de lissage.

Une autre option consiste à exploiter les fonctionnalités de création et d'édition de lignes, et de tracer directement soi même ses propres lignes sur l'oreille intacte, puis de s'en servir ensuite comme base pour notre traitement. On peut également directement tracer une ligne d'édition sur

le patch et le déformer manuellement (figure 23).



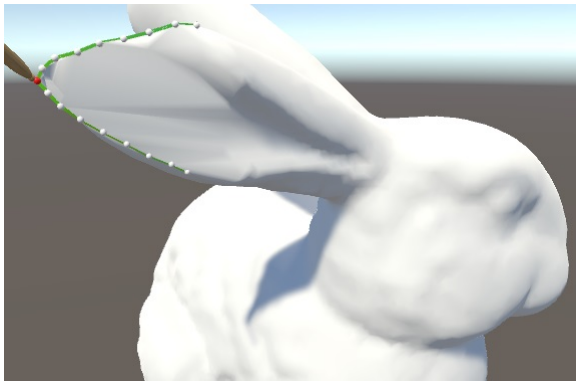


FIGURE 23 – Définition des lignes de contrainte par symétrie à la main (image de gauche) ou déformation directe par édition d'une ligne d'édition (image de droite).

6.3 Apport de l'immersion Virtuelle

Afin d'évaluer au mieux l'apport de l'immersion virtuelle, nous avons prévu de réaliser une expérimentation, qui consistera à faire tester les différentes fonctionnalités de l'application à un certain nombre de personnes, et à les leur faire comparer à la réalisation de ces actions dans un environnement classique de bureau. A cette fin, une alternative à la souris et au clavier a été développée pour les principales fonctionnalités de l'application (le flux de traitement principal : calcul/sélection des ridges, symétrie, et projection des contraintes de position pour la déformation). Nous pourrons ainsi évaluer l'apport de la réalité virtuelle sur des critères objectifs (temps de réalisation des tâches par l'opérateur,...) et d'autres plus subjectifs (élaboration d'un questionnaire pour recueillir les impressions). Cette expérimentation devrait avoir lieu courant Septembre aussi ne pouvons-nous pas commenter ses résultats pour l'instant, mais nous pouvons déjà faire quelques remarques.

Nous allons dans un premier temps parler de l'apport du bras haptique. Si l'on se sert de ce dernier pour réaliser la quasi-totalité des actions présentées, à quel moment le retour de force représente t-il un réel apport ?

Par rapport à un autre dispositif de contrôle en immersion virtuelle (souris 3D, contrôleur du casque RV Vive), le fait d'avoir un retour de force est très pratique pour toutes les tâches de base qui demandent un peu de précision (sélection d'une ligne, d'un point sur la surface). On pourrait cependant réaliser ces actions assez simplement à la souris.

Là où les dispositifs de contrôle en 3D présentent un réel avantage par rapport à une souris dans un environnement de bureau, c'est lorsqu'il s'agit de déplacer des objets dans l'espace 3D, comme lorsque l'on déplace le plan pour la symétrie ou la projection. Dans ces deux cas cependant le retour de force ne parait pas être un apport indispensable. Pour l'édition des lignes sur la surface du maillage, il est par contre très pratique.

Mais il représente surtout un réel apport pour les techniques alternatives présentées par la figure 23, lorsque l'on souhaite tracer directement sur le maillage ou le déformer interactivement. Cependant, il est plus difficile que nous ne l'imaginions au départ de suivre les formes caractéristiques du maillage avec le stylet du bras haptique. Si il est par exemple facile de suivre un creux, il sera bien plus délicat de suivre une bordure à forte courbure positive.

Enfin, nous ne pouvons pas encore vraiment discuter de l'apport de la visualisation en 3D pour réaliser les actions de l'application car pour des raisons matérielles, nous n'avons pas encore pu brancher en même temps le casque RV et le bras haptique, mais cela sera bientôt le cas, et l'expérimentation donnera une vision plus large de l'apport que représente l'immersion virtuelle.

6.4 Limites et améliorations envisageables

Pour finir ce rapport, nous allons revenir sur les différentes limites de la solution mise en œuvre. Certaines ont déjà été évoquées tout au long de ce rapport, d'autres vont être mise en évidence

dans cette section. L'objectif est de faire ressortir quelques axes majeurs intéressants à approfondir par la suite.

- D'abord, la chaîne de traitement et les solutions retenues dans cette mise en œuvre ne permettent de travailler que sur des objets qui présentent des symétries planaires à exploiter, mis à part pour la déformation directe du patch qui peut s'appliquer dans n'importe quelles conditions, mais là encore la solution implémentée dans l'application est très basique puisqu'elle ne permet de manipuler qu'une ligne d'édition à la fois, et mériterait peut-être d'être également développée. En tout cas il serait intéressant de réfléchir à d'autres moyens de placer à notre convenance des motifs obtenus par calcul des ridges, de manière plus libre mais tout aussi intuitive.
- Le problème de l'appariement mérite également d'être creusé. La solution retenue actuellement, qui consiste à projeter les lignes de contrainte sur le patch dans une direction orthogonale à un plan, limite l'utilisation de la méthode à des objets dont la partie formée de la zone reconstruite et de la zone symétrique servant de base à la reconstruction est convexe. Dans le cas contraire certaines contraintes ne pourront pas être projetées sur la surface, soit parce que leur projection sera hors de la zone du patch, soit parce qu'elles seront alignées avec d'autres positions de contrainte dans la direction de projection. Il faudrait donc réfléchir à un autre type de projection, ou éventuellement à une procédure de projection en plusieurs étapes. Ou également envisager un autre algorithme pour faire l'appariement.
- Un autre problème, d'ordre tout à fait technique cette fois, est lié au fonctionnement d'Unity et à la manière dont est implémentée la solution. Il est actuellement impossible de traiter dans l'application des maillages de trop grande taille. En effet, les traitements réalisés actuellement par les scripts implémentés vont directement opérer sur le composant *MeshFilter* dans lequel le maillage est stocké. Or, lorsque l'on charge des maillages comptant plus de 65534 sommets ou 65534 facettes, Unity va créer automatiquement plusieurs sous-objets pour stocker le maillage dans plusieurs *MeshFilter* pour des raisons d'optimisation. Si l'on veut pouvoir charger de plus gros maillages, il faudrait donc modifier les scripts qui manipulent le maillage pour les adapter à cette architecture.
- On pourrait également revenir sur l'opération de calcul des ridges. Si l'on souhaite accélérer le temps de calcul pour pouvoir régler les paramètres de filtrage de façon plus interactive, il serait intéressant d'être en mesure de limiter le calcul des ridges à une zone d'intérêt sélectionnée.
- Enfin, comme nous l'avons signalé au chapitre précédent, l'opération de déformation du maillage prend trop de temps pour pouvoir être réalisée en temps réel. Sur un maillage comptant 9157 sommets et 18164 facettes, cette dernière dure environ 600ms. C'est trop pour du temps réel. Pourtant, en théorie, les fonctions fournies par le package CGAL permettent le temps réel. L'explication est la suivante :
  L'opération de déformation est basée sur la résolution d'un système linéaire, tel que décrit dans l'équation 2 de la section 5.3.2. Or dans ce système, la matrice de la partie à gauche du signe égal ne dépend que du maillage initial et de quels sont les sommets qui sont des sommets de contrôle de la déformation. Cette partie est factorisée avant la résolution du système. Cette factorisation peut être réutilisée lors des résolutions successives et donc des déformation successives tant que l'on ne change pas les sommets de contrôle, et le temps d'exécution est alors moindre. Cependant dans notre implémentation, à chaque appel de la fonction de déformation de la dll par l'application Unity3D, tout le maillage est passé en paramètre, reconstruit dans une représentation utilisant les demi-arêtes et tout le système de déformation est également reconstruit, que les points de contrôle soient les mêmes ou non. Si nous pouvions garder le système de représentation en demi-arête du maillage actif, ainsi que le système de déformation, nous pourrions probablement diminuer le temps d'exécution pour réaliser la déformation en temps réel. A cette fin, il faudrait exécuter les fonctions de déformation non plus dans une librairie mais dans un programme exécuté en parallèle et qui communiquerait avec Unity3D pour lui envoyer les informations sans que l'on soit obligé de reconstruire le maillage à partir de zéro pour la moindre opération. Une évolution intéressante pourrait donc être une refonte de l'architecture du projet pour permettre cette exécution en parallèle.

7 Conclusion

En l'état actuel, le projet apporte une solution nouvelle à la problématique de complétion de maillages présentant des zones fortement endommagées, pour lesquelles un simple lissage ne donne pas un résultat satisfaisant. L'utilisation de simples contraintes de position pour la reconstruction donne des résultats encourageants, cependant nous aimerions avoir des résultats plus précis pour le calcul des ridges, qui fassent apparaître de manière vraiment plus nette les formes significatives du maillage, afin que l'on soit capable de reconstruire même des motifs assez fins. A mon avis, la recherche d'idées concernant la construction des contraintes est à poursuivre donc.

D'un point de vue personnel, ce stage m'a apporté des compétences et du savoir-faire à plusieurs niveaux. D'une part des compétences techniques. Le fait de concevoir et de réaliser un projet de A à Z est très formateur ; j'ai pu apprendre à utiliser des outils informatiques comme Cmake, ou git pour gérer les versions de mon projet qui est hébergé sur la forge du Liris. J'ai également pu apprendre à utiliser des fonctionnalités assez avancées du moteur de jeu Unity3D, à utiliser une librairie de géométrie utilisant de nombreux "templates" comme CGAL.

D'autre part, des compétences liées à l'analyse et à la recherche. Le fait de travailler dans un contexte de recherche, abordant des idées novatrices et utilisant des outils modernes est particulièrement intéressant. Cela m'a permis de mieux comprendre l'enjeu d'un travail de recherche scientifique, qui impose une certaine rigueur dans la compréhension des outils utilisés et de l'environnement dans lequel on évolue afin d'être capable de proposer à son tour des idées intéressantes. La recherche et la proposition de nouvelles idées est d'ailleurs à la fois la principale difficulté et un des intérêts principaux de ce type de travail.

8 Annexes

A

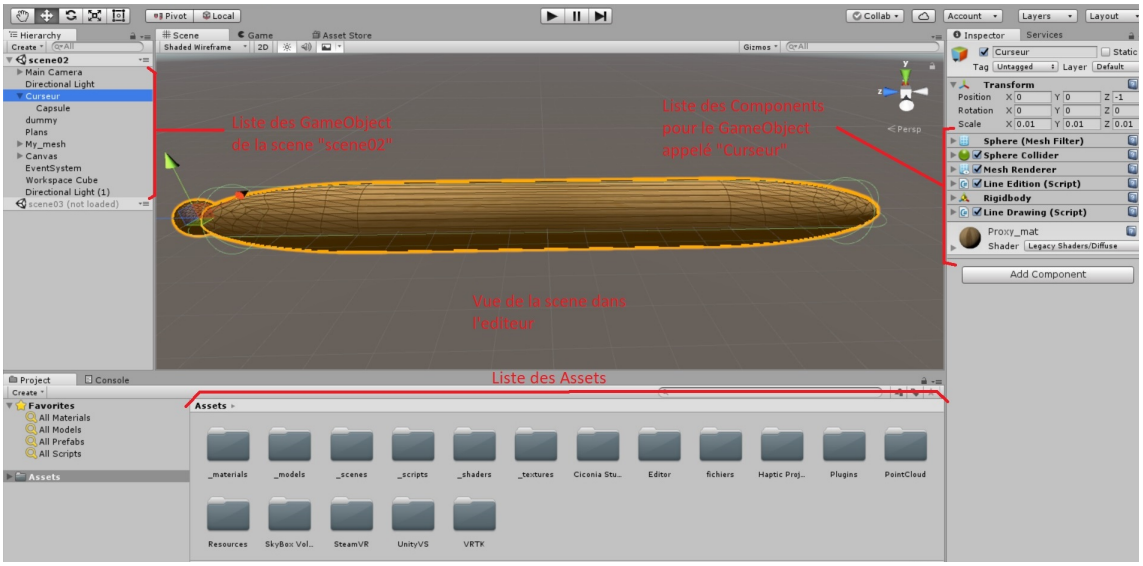


FIGURE 24 – Interface de l'éditeur d'Unity3D

B

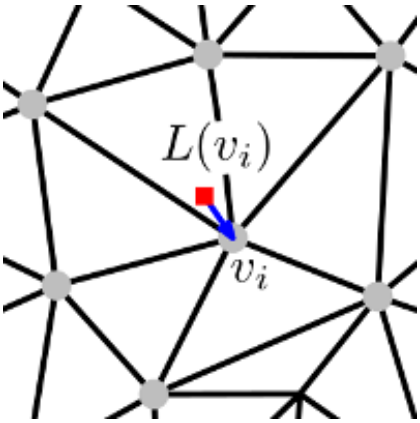


FIGURE 25 – représentation Laplacienne du sommet Vi avec poids uniformes. Le point rouge est le barycentre. Image issue de [11]

C

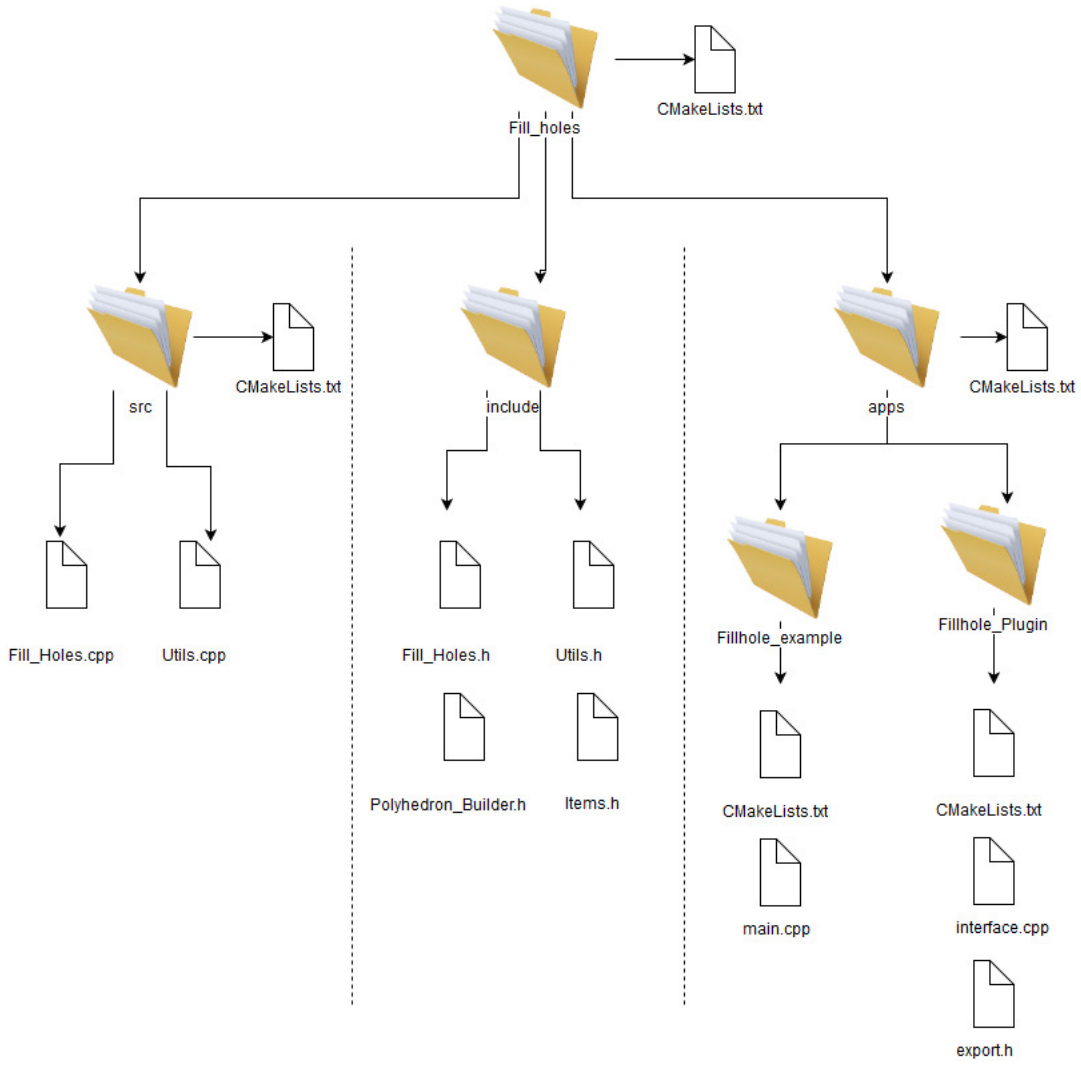


FIGURE 26 – structure du projet C++ permettant de compiler la librairie fillholePlugin.dll importée sous Unity3D

D

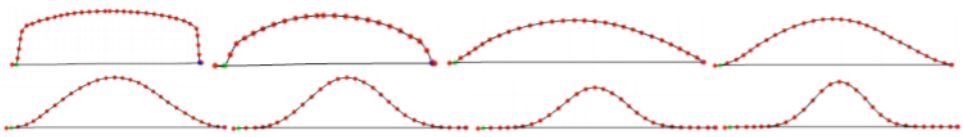


FIGURE 27 – illustration de la forme de la courbe déformée pour les valeurs de α suivantes : 0.1,0.25,0.5,0.75,1,2,3,4. image issue de [15]

references

[1] **P. Liepa**, 2003, Filling holes in meshes, Symposium on Geometry Processing, pages 200–207.

[2] **Pernot J-P., Moraru G., Véron P.**, (2007), Repairing triangle meshes built from scanned point cloud, Journal of Engineering Design, vol. 18(5), pp.459-73.

[3] **M.Botsch and O.Sorkine**. On linear variational surface deformation methods. IEEE Transactions on Visualization and Computer Graphics, 14(1) :213–230, 2008.

[4] **Lutz Kettner** . 3D Polyhedral Surface. InCGAL User and Reference Manual. CGAL Editorial Board, 4.10 edition, 2017.

[5] **Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy**. Polygon Mesh Processing, chapter 1,6,8. AK Peters / CRC Press, 2010.

[6] **Marc Pouget and Frédéric Cazals** . Approximation of Ridges and Umbilics on Triangulated Surface Meshes. In CGAL User and Reference Manual. CGAL Editorial Board, 4.10 edition, 2017.

[7] **F. Cazals, M. Pouget**, Estimating differential quantities using polynomial fitting of osculating jets. Computer Aided Geometric Design, Volume 22(2), 2005, pp 121-146.

[8] Topology driven algorithms for ridge extraction on meshes. **Cazals, Pouget.** 2005.

[9] Ridge-valley lines on meshes via implicit surface fitting. **Ohtake, Belyaev, Seidel.** 2004.

[10] **Sébastien Loriot, Jane Tournois, Ilker O. Yaz.** Polygon Mesh Processing. InCGAL User and Reference Manual. CGAL Editorial Board, 4.10 edition, 2017.)

[11] **Sébastien Loriot, Olga Sorkine-Hornung, Yin Xu and Ilker O. Yaz.** Triangulated Surface Mesh Deformation. In CGAL User and Reference Manual. CGAL Editorial Board, 4.10 edition, 2017.

[12] **Olga Sorkine and Marc Alexa**. As-rigid-as-possible surface modeling. In ACM International Conference Proceeding Series, volume 257, pages 109–116. Citeseer, 2007.

[13] **Isaac Chao, Ulrich Pinkall, Patrick Sanan, and Peter Schröder**. A simple geometric model for elastic deformations. In ACM SIGGRAPH 2010 papers, SIGGRAPH '10, pages 38 :1–38 :6. ACM, 2010.

[14] **Zohar Levi and Craig Gotsman**. Smooth rotation enhanced as-rigid-as-possible mesh animation. IEEE Transactions on Visualization & Computer Graphics, 2015.

[15] **Manolya Eyiyurekli and David Breen**. Localized Editing of Catmull-Rom Splines. Computer-Aided Design and Applications, 2009.

[16] **Zhifan Jiang, Fabrice Jaillet, Florence Zara**. Reconstruction et complétion de maillages sous contraintes. [Rapport de recherche] LIRIS UMR CNRS 5205. 2011

[17] **Armand Le Gouguec, Fabrice Jaillet, Ruding Lou**, Complétion de maillage 3D par analyse sémantique. [Rapport de Projet de Fin d'Études] Arts et Métiers ParisTech Cluny, 2014.