Freestyle : Sculpting Meshes with Self-Adaptive Topology

Rapport Conception

Étudiants : Charles Garibal, Maxime Robinot, Mathieu Dachy
Tuteur : Loïc Barthe
09/01/2015

Freestyle : Sculpting Meshes with Self-Adaptive Topology
I. Introduction
II. Vue d’ensemble détaillée du système
- Cas d’utilisation
- Quelques diagrammes de séquence
  - Ouvrir un fichier objet 3D
  - Visualiser les paramètres du maillage
  - Choisir un opérateur
  - Appliquer un opérateur

III. Diagramme de classe
- Freestyle IHM
- “Sculptor”

IV. Mise à jour des prévisions
V. Conclusion

I. Introduction

Le but de notre chef-d'oeuvre est de comprendre et d'implémenter au sein d'un moteur graphique des algorithmes issus de l'article de recherche *Freestyle : Sculpting Meshes with Self-Adaptative Topology*. Cette publication propose une nouvelle approche de modélisation 3D à travers la manipulation d'outils intuitifs permettant à l'utilisateur d'appliquer des déformations sur un objet. Ainsi, même si ce genre d'application est reservée aux infographistes, un néophyte pourrait modifier l'apparence d'un objet à sa guise.

D'un point de vue technique, l'utilisateur manipulera un maillage particulier dit "quasi-uniforme" et pourra modifier sa topologie en formant des trous ou en fusionnant des régions. La complexité d'un tel module réside dans le contrôle et la modification de la structure du maillage, la détection et le traitement des changements de topologie, l'implémentation des outils de déformations ainsi que la conception de l'interface qui permettra de les utiliser.

II. Vue d'ensemble détaillée du système

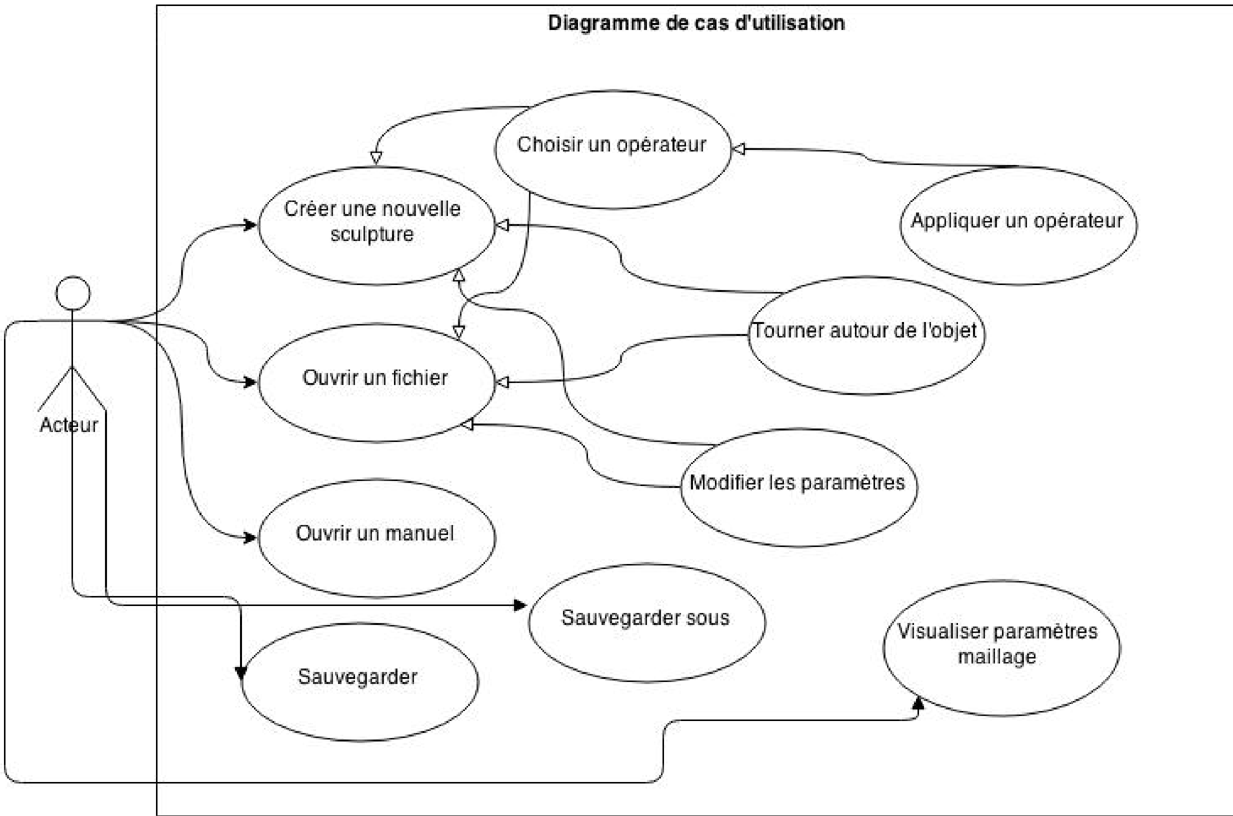
Notre application s'organise autour de trois grands modules, le moteur graphique (vortexengine) que nous avons utilisé durant cette année, l'application, qui contient l'IHM et notre module de sculpture que nous avons appelé "Sculptor".

Notre principal but est de fournir un code applicatif simple qui démontre le fonctionnement de notre module. L'architecture de l'application ne repose pas à proprement parlé sur un patron MVC qui ne ferait qu'alourdir inutilement le code. L'application communique avec notre module par le biais de la classe "Sculptor", nous avons souhaité que ce module soit indépendant de l'application afin qu'il puisse être réutilisé dans n'importe quel projet utilisant OpenMesh.

Tout l'enjeu est de fournir à notre framework le plus de fonctionnalités génériques possibles pour garantir un environnement stable au futur programmeur.

Afin de définir les classes nécessaires à notre application, nous avons commencé par lister les différents cas d'utilisation des utilisateurs. On a supposé qu'il n'y a qu'un type d'utilisateur.

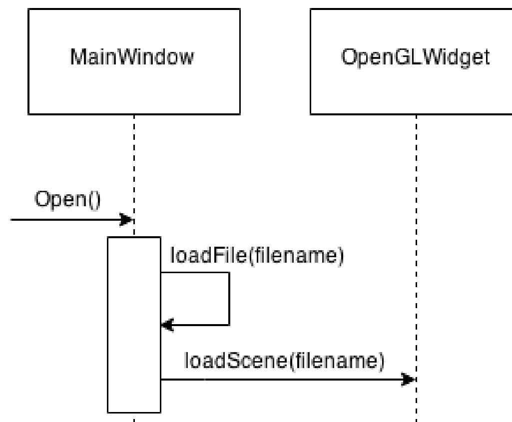
A. Cas d'utilisation



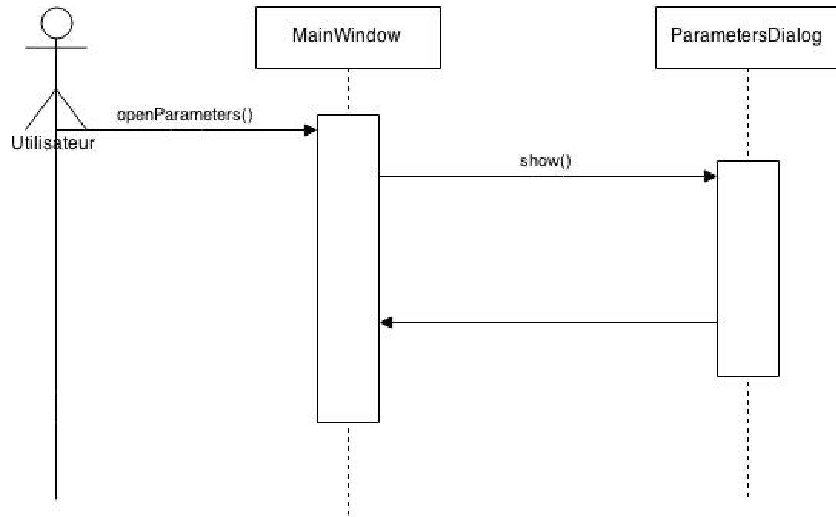
Nous allons maintenant détailler chacun des cas d'utilisations avec un diagramme de collaboration. De ces diagrammes découleront les classes nécessaires à notre projet.

B. Quelques diagrammes de séquence

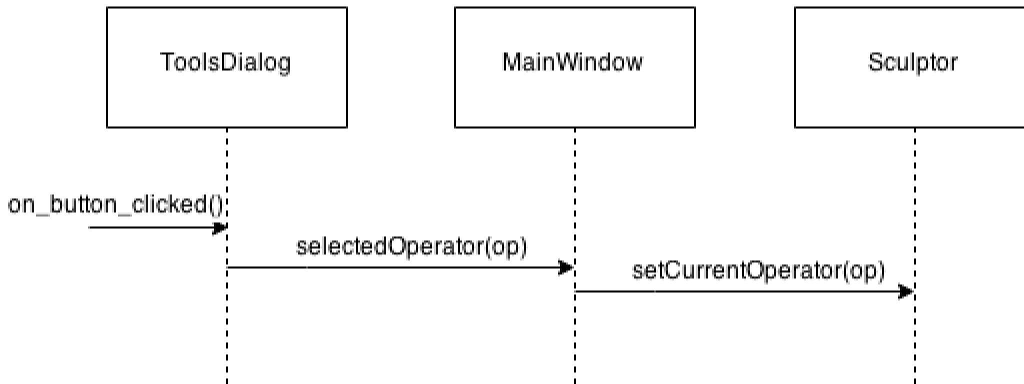
Ouvrir un fichier objet 3D



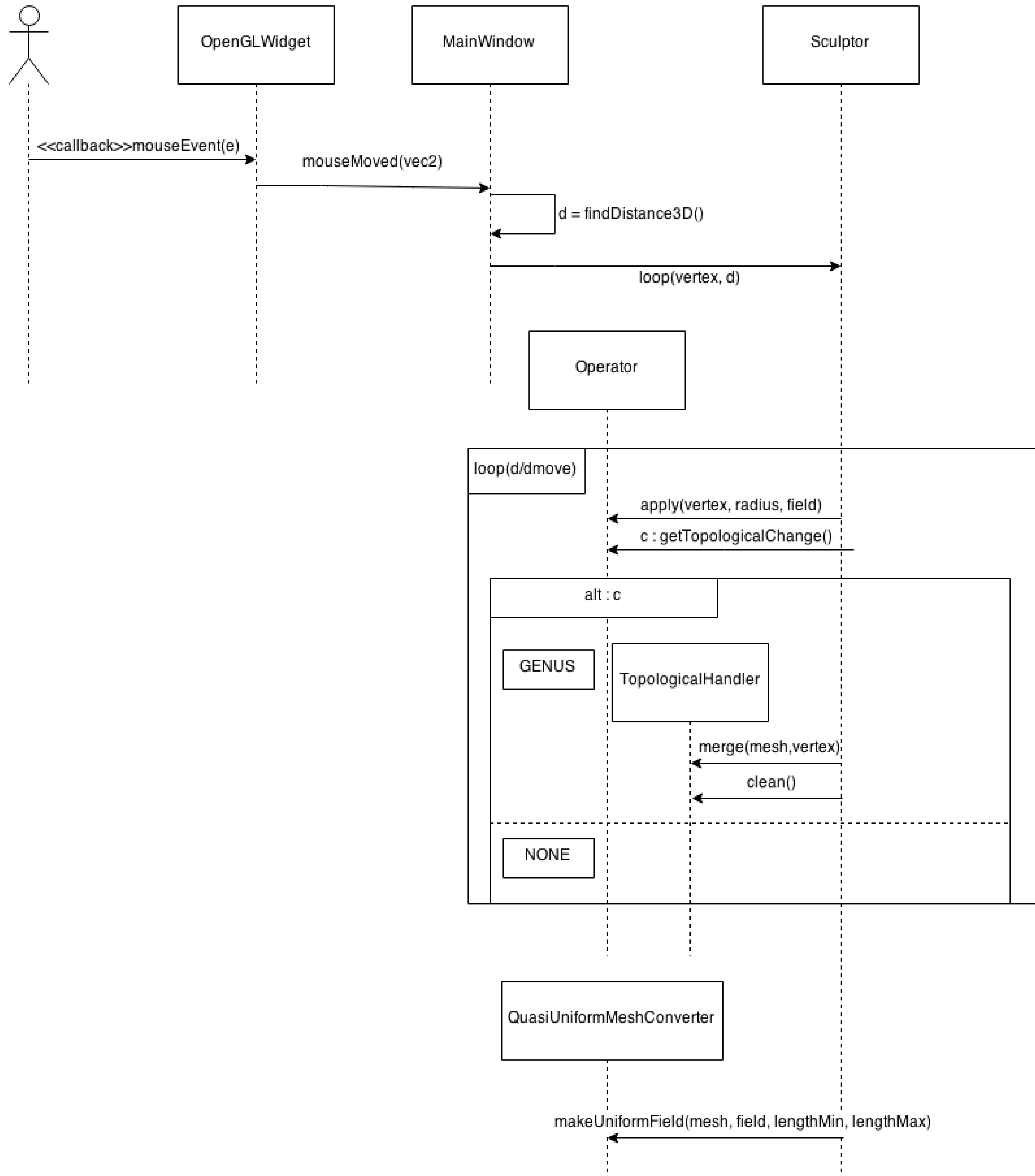
Visualiser les paramètres du maillage



Choisir un opérateur

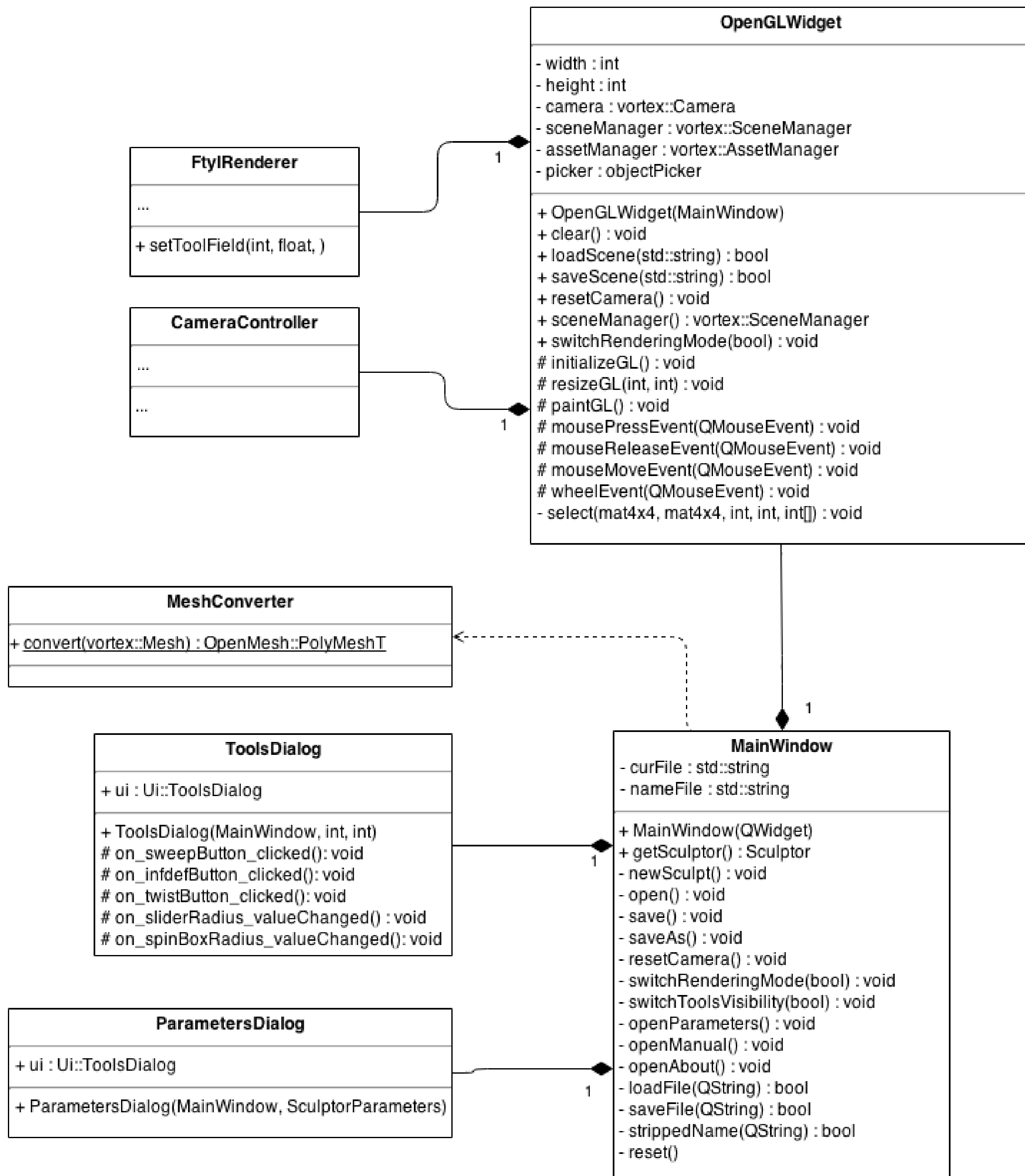


Appliquer un opérateur

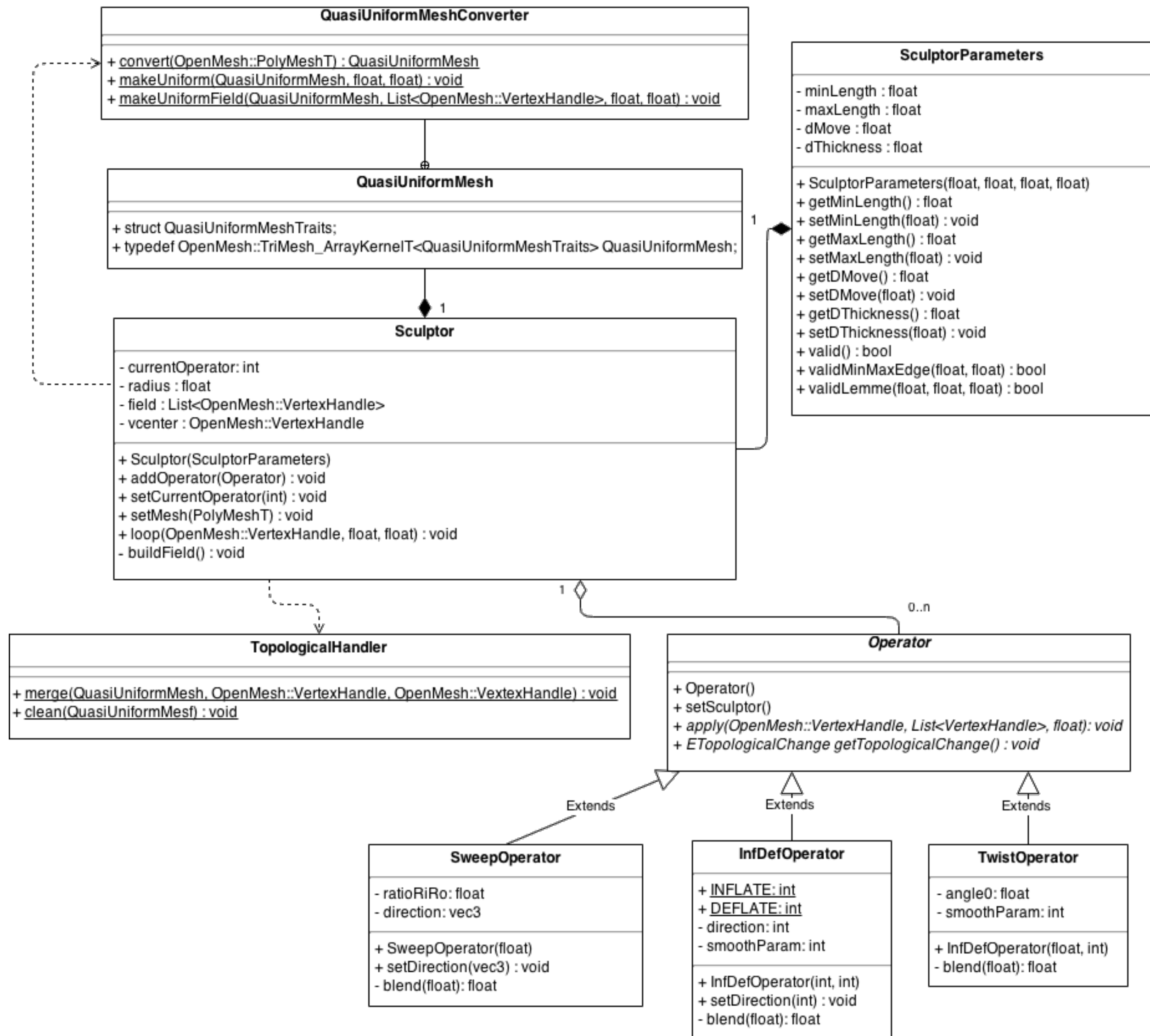


III. Diagramme de classe

A. Freestyle IHM



B. "Sculptor"



Classe QuasiUniformMesh:
C'est une représentation d'un maillage OpenMesh triangulaire manifold qui respecte les contraintes d'un maillage quasi uniforme. Ce dernier est un maillage qui à une taille maximale et minimale de longueur d'arêtes. Cette classe contient la classe interne QuasiUniformMeshConverter qui contient des fonctions static de conversion. La fonction convert prend un mesh OpenMesh quelconque en paramètre et le convertit en QuasiUniformMesh triangulaire manifold. Ensuite la fonction makeUniform rend le maillage quasi uniforme selon les paramètres. La fonction makeUniformField rend quasi uniforme la région dans la zone d'influence.

Classe TopologicalHandler:
Cette classe permet de gérer les changements de topologies du maillage quasi uniforme, elle possède une méthode *"merge"* qui gère la fusion de deux sommets non adjacents et proche de de *dthickness*. Elle possède également la fonction "*clean*" qui a pour but d'éliminer des parties indésirables tel que des paires de triangles qui ont les mêmes sommets ou encore de séparer la surface lorsque deux arêtes coïncident.

Classe SculptorParameters:
Cette classe contient les métriques permettant de paramétrer un maillage quasi-uniforme. Cette classe permet notamment de vérifier les contraintes associés à ce type de maillage, comme par exemple avec la méthode "*validLemme"* qui permet de vérifier la propriété suivante :

$$dmove \leq (dthickness - ddetail/\sqrt{3}) / 2$$

Elle permet notamment de savoir pour à partir de quelle distance *dthickness* entre deux sommets non adjacents il est nécessaire de déclencher une fusion.

Classe Operator:
Cette classe décrit le fonctionnement générique d'un opérateur de déformation de maillage. La transformation s'effectue sur l'ensemble des sommets contenus dans la sphère ayant pour centre un sommet du maillage et un rayon fixé par l'utilisateur. La classe mère, le sculpteur, lui donne en complément la norme du mouvement à effectuer. Si un opérateur doit prendre des paramètres supplémentaires pour effectuer la déformation ceux ci devront être mis en place par l'application. Ces paramètres sont le plus souvent fixés en fonction des actions de l'utilisateur (mouvement de souris). Ce choix de conception permet de créer facilement de nouveaux opérateurs qui fonctionnent sur le même principe.

Un opérateur doit pouvoir dire si sa transformation entraîne un changement de topologie pour que le sculpteur puisse garantir la validité du maillage, c'est le rôle de la fonction getTopologicalChange().

Nous avons choisis d'implémenter trois opérateurs, voici leurs spécificités :

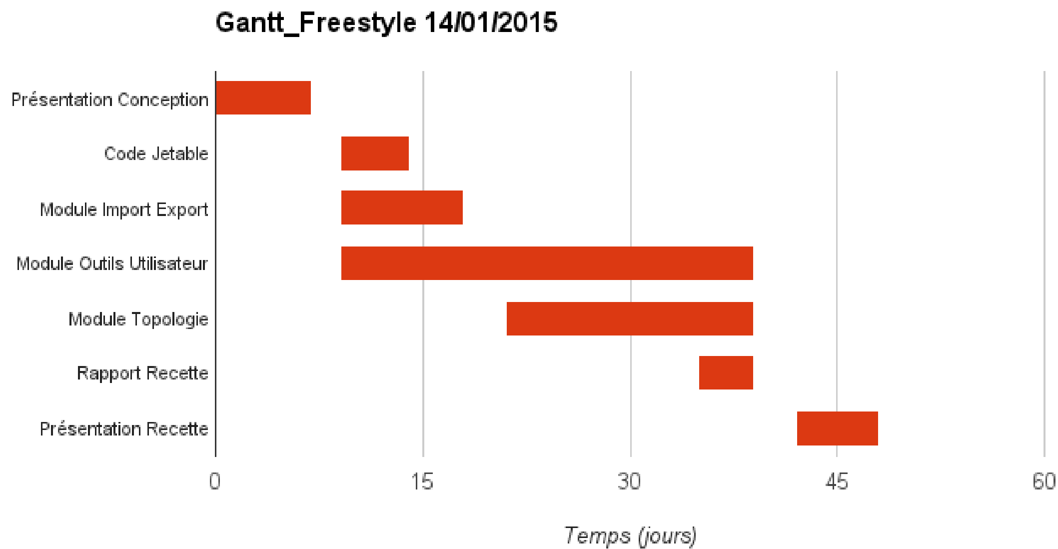
- Classe SweepOperator : Représente l'opération "Etendre" qui a la particularité d'être dépendante d'un vecteur arbitraire, celui ci doit donc être fixé à posteriori avant chaque déformation. La déformation s'effectue dans deux "sous-zones", le ratioRiRo permet de calculer automatiquement ces zones en fonction du rayon de la sphère.
- Classe InfDefOperator : Représente l'opération "Bomber/Creuser" qui comprend un paramètre global supplémentaire permettant de contrôler la continuité aux bords de la déformation. L'application décide du sens de la déformation grâce au paramètre direction.
- Class TwistOperator : Représente l'opération "Tordre". Comment spécifié dans la publication, elle possède le même paramètre global que "Bomber/Creuser" et en possède un de plus pour contrôler la "puissance" de la rotation.

<u>Classe Sculptor:</u>
La classe Sculptor est le coeur du framework QuasiUniform. C'est elle qui gère le "sampling" du mouvement utilisateur et qui fait les appels aux opérateurs, au topologicalHandler et aux fonctions d'uniformisation du maillage. L'application s'occupe de créer des instances de chaque opérateurs mais elle doit les passer au Sculptor via la fonction addOperator. La fonction setCurrentOperator permet à l'interface de mettre à jour l'opérateur sélectionné par l'utilisateur.

Après la création du Sculptor, l'application doit lui donner un maillage OpenMesh quelconque par la fonction setMesh. Cette fonction va appeler la fonction static de conversion convert située dans une classe interne à la classe QuasiUniformMesh pour rendre le maillage triangulaire. Ensuite elle appellera la fonction makeUniform située au même endroit pour uniformiser le maillage.

La fonction loop est appelée à chaque fois que l'utilisateur veut appliquer un opérateur. Elle prend en paramètre le sommet cliqué, le rayon de la sphère influente, la longueur du mouvement et appel la fonction buildField qui détermine la région qui est dans la zone d'influence. Si le mouvement est plus grand que le paramètre d*move*, alors il est divisé en plusieurs sous-mouvements inférieur ou égaux à d*move*. Ensuite elle boucle sur le nombre de sous-mouvements, en appelant la fonction apply de l'opérateur courant, puis les fonctions du topologicalHandler si besoin, et enfin la fonction makeUniformField qui uniformise la région influencée.

IV. Mise à jour des prévisions



Concernant la planification, il n'y a pas eus de modification quand à nos prévisions pour la réalisation des différents modules, les dates de fin de développement pour chaque partie du travail sont donc maintenues.

V. Conclusion

Nous avons actuellement bien avancé dans la construction de l'architecture de notre application, nous avons détaillé dans les différents diagrammes de séquence et les diagrammes de classes aussi bien pour l'interface homme-machine que pour le coeur du logiciel Freestyle (Sculptor). Nous avons donc une bonne idée de la façon dont notre logiciel fonctionne au niveau comportemental, bien qu'il est toujours possible que les choses évoluent par la suite (entre autres sur la souplesse des paramètres du maillage).

Nous n'avons pas souhaité modifier notre planification puisque nous ne pouvons pas réellement affirmer que nous pourrions finir les modules plus ou moins vite que prévu, mais au vue de la conception et de l'architecture générale nous pensons être dans les temps.